AIR Project Final Report

José Rufino (DI-FCUL)
Sérgio Filipe (Skysoft Portugal, SA.)

DI-FCUL TR–07–35

December 2007

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749-016 Lisboa
Portugal

Abstract

This document describes the main results of AIR, an innovation initiative sponsored by ESA, the European Space Agency. The acronym AIR stands for ARINC 653 Interface in RTEMS. The ARINC 653 is a civil aviation world specification addressing safety critical and certification issues in embedded systems software. The AIR Project studied the adoption of ARINC 653 in space on-board software together with the utilization of RTEMS, the Real-Time Executive for Multiprocessor Systems.

This document: (i) describes the main issues regarding the AIR architecture specification; (ii) addresses how space and time partitioning could be provided in an abstract processor infrastructure, as well as those requirements can be mapped into both SPARC ERC32/LEON and Intel IA-32 (80x86) architectures; (iii) describes how to achieve the mapping of the ARINC 653 service interface in RTEMS; (iv) identifies the most relevant module dependencies of RTEMS with regard to AIR implementations; (v) identifies a preliminary set of modifications to be introduced in the RTEMS application production chain for the implementation of AIR-based systems (exemplified through a proof of concept prototype).

AIR

ARINC 653 Interface in RTEMS

AIR Final Report

	Name	Title	Signature	Date
Written	José Rufino	Senior Researcher		12-11-2007
Written	Sérgio Filipe	Technical Manager		12-11-2007
Verified	José Rufino	Senior Researcher		12-11-2007
Verified	Sérgio Filipe	Technical Manager		12-11-2007
Verified	José Neves	Project Manager		12-11-2007
Approved	José Neves	Project Manager		12-11-2007

Issue	Date	Description	Author
1.0	12-11-2007	ARINC 653 Interface in RTEMS – Final Report	Sérgio Filipe, José Rufino

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

The purpose of this document is to describe the study and results of the activities performed on the AIR (ARINC 653 Interface in RTEMS) project.

1.2 Scope

This document defines the AIR Project WP3 output: AIR Final Report.

1.3 Applicable Documents

Reference	Title
[AIRPROP]	AIR Proposal, Skysoft, September 2005
[TMREPLY]	Technical Memo for the AIR Proposal (negotiation points replies)
[AIRMMIN]	Minutes of Meeting (From Project's Kick-Off Meeting to the Sixth Internal Meeting)
[AIRWP1]	AIR WP1 Output: Requirements, Architecture and Services
[AIRWP2]	AIR WP2 Output: Overall System Specification
[AIR WP3]	AIR WP3 Output: AIR Design Results and Proof of Concept

Table 1 – Applicable Documents

1.4 Reference Documents

Reference	Title
[A653P1]	ARINC Specification 653 Part 1 (Supplement 2), Required Services, Aeronautical Radio Inc., 2006
[A653P2]	ARINC Specification 653 Part 2, Extended Services DRAFT 5, Aeronautical Radio Inc., 2006
[RUFDIN]	N. Diniz, J. Rufino, ARINC 653 In Space, Dasia 2005, Eurospace, Edinburgh
[RUSHB1]	J. Rushby, Partitioning in Avionics Architectures: Requirements, Mechanisms, and Assurance, Technical Report NASA / CR-1999-209347, SRI International, 1999
[DO248B]	RTCA DO-248B, Final Annual Report For Clarification Of DO-178B "Software Considerations In Airborne Systems And Equipment Certification"
[RTEMSQUAL]	J. Seronie-Vivien, C. Cantenot, RTEMS Operating System Qualification for the ERC32 Target, DASIA 2005, Eurospace Edinburgh
[RTEMSAPI]	RTEMS C Users Guide – Native API, Edition 4.6.6, for RTEMS 4.6.6 - OAR
[RTEMSPOSIX]	RTEMS POSIX API User's Guide - Edition 4.6.6, for RTEMS 4.6.6 - OAR
[RTEMSI386]	RTEMS Intel i386 Applications Supplement - Edition 4.6.6, for RTEMS 4.6.6 - OAR
[RTEMSSPARC]	RTEMS SPARC Applications Supplement - Edition 4.6.6, for RTEMS 4.6.6 - OAR
[COUT05a]	M. Coutinho, J. Rufino, C. Almeida. Control of Event Handling Timeliness in RTEMS. Proceedings of the 17th IAESTED International Conference Parallel and Distributed Computing Systems, Phoenix, USA, November 2005.
[COUT06a]	M. Coutinho, C. Almeida, J. Rufino. VITRAL - A Text Mode Window Manager for Real-Time Embedded Kernels. Proc. 11th IEEE Int. Conf. Emerging Technologies Factory Automation, Prague, Czech Republic, September, 2006.

Table 2 – Reference Documents

1.5 Document Outline

Section 1	Introduces the document by indicating its purpose and scope, its content and which documents it refers to.
Section 2	Introduces the AIR fundamental concepts.
Section 3	Describes the AIR architecture specification.
Section 4	Addresses the fundamental mechanisms for securing spatial and temporal segregation.
Section 5	Describes the interactions between the APEX interface and the AIR Partition Management Kernel (PMK)
Section 6	Describes the RTEMS Module Dependency in the implementation of AIR-based systems and applications.
Section 7	Addresses details of AIR applications production tool chain and its relation with canonical development environments.
Section 8	Provides an overview of the AIR proof of concept prototypes
Section 9	Concludes the document.
Annex A	Summarizes a set of AIR Project Result indicators.
Annex B	Presents the APEX Interface Demonstrator Activity Diagrams

1.6 Document Contributions

This document was jointly prepared by Faculdade de Ciências da Universidade de Lisboa (FCUL) and Skysoft Portugal.

1.7 Acronyms and Abbreviations

Acronym	Definition
AEEC	Airlines Electronic Engineering Committee
AIR	ARINC 653 Interface in RTEMS
APEX	Application Executive
ARINC	Aeronautical Radio, Inc.
API	Application Programming Interface
BSP	Board Support Package
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
DDD	Data Display Debugger
eCos	Embedded Configurable Operating System
EUROCAE	European Organization for Civil Aviation Equipment
FAA	Federal Aviation Administration
FCUL	Faculdade de Ciências da Universidade de Lisboa
FIFO	First In First Out
GCD	Greatest Common Divisor
GDB	GNU Debugger
GNU	GNU's Not Unix
GRUB	GRand Unified Bootloader
HAL	Hardware Abstraction Layer
IMA	Integrated Modular Avionics
ISR	Interrupt Service Routine
MEC	Multi-Executive Core
MILS	Multiple Independent Levels of Security
MMU	Memory Management Unit
MTF	Major Time Frame
OS	Operating System
PMK	Partition Management Kernel
PST	Partition Scheduling Table
PTD	Page Table Descriptor
PTE	Page Table Entry
RAM	Random Access Memory
RTCA	Radio Technical Commission for Aeronautics
RTEMS	Real-Time Executive for Multiprocessor Systems
RTOS	Real-time Operating System
SEC	Single-Executive Core
SIS	SPARC Instruction Simulator
TCB	Task Control Block

Table 3 – Acronyms and Abbreviations

2 AIR Fundamental Concepts

2.1 Introduction

In both avionics and space industries, the safety concept is of paramount importance. The ARINC 653 standard was developed with the purpose that all safety critical software embedded in a system must follow very strict and demanding rules both in terms of operation and certification.

ARINC 653 and Integrated Modular Avionics (IMA) are the answers provided by the civil aviation world to problems that are also identified in the space world. The space world is looking for a standardized interface for the Operating Systems (OS) located on board the spacecrafts. Most of the requirements from the civil aviation world that led to the definition of ARINC 653 are also requirements from the space world and thus the adaptation of the specification to the space world needs can be performed with minor changes, keeping its basic principles.

The adoption of the **ARINC 653 concept in space on-board software** will not only provide the space industry the same benefits the aviation industry has already profited with by adopting the standard – software portability and modularity, partitioning and less certification effort, etc. It will also promote the reusability of Research and Development (R&D) efforts already invested in the scope of another industry domain, further increase the synergies in the development of software for the parallel domains of civil aviation and space and potentiate reduction in the development costs of on-board software. Finally, the space world will benefit from ARINC 653's improvement in the development framework available for both application developers and integrators.

Furthermore, there is a general demand for the use and re-utilization of commercial off-the-shelf (COTS) components in the design of complex embedded systems, such as those found in aerospace applications. The AIR – ARINC 653 Interface in RTEMS – innovation initiative has emerged complying to this requirement, exploiting the utilization of a COTS licence-free open-source real-time operating system, the Real-Time Executive for Multiprocessor Systems (RTEMS). The use of RTEMS is particularly interesting given its qualification for critical on-board software of unmanned space programs.

The present document summarizes the main AIR design results: (i) it describes the AIR architecture specification; (ii) addresses how space and time partitioning could be provided in an abstract processor infrastructure, as well as those requirements can be mapped into both SPARC ERC32/LEON and Intel IA-32 (80x86) architectures; (iii) describes how to achieve the mapping of the ARINC 653 service interface in RTEMS; (iv) identifies the most relevant module dependencies of RTEMS with regard to AIR implementations; (v) identifies a preliminary set of modifications to be introduced in the RTEMS application production chain for the implementation of AIR-based systems (exemplified through a proof of concept prototype).

2.2 ARINC 653 Standard

The ARINC 653 standard in essence defines an Application Executive (APEX) interface between the Operating System (OS) of a safety critical system and the application software developed for that system. The standard specification defines how data may be exchanged statically (through a given configuration profile) or dynamically (via the use of services). The ARINC 653 standard specifies the behavior of the set of services provided by the OS and used by the application.

The runtime environment provided by the APEX interface enables independently produced applications to execute together in the same hardware, providing support to spatial and time segregation of these applications through the usage of **partitions**.

The ARINC 653 standard specification aims language and hardware independency, thus promoting portability of the software whilst allowing modularity and versatility in the design, and flexibility in the selection of development tools, host platforms and target infrastructures. The standard specification also establishes a strict control over a computational system, both in terms of temporal and spatial segregation, as well as in the provision of a standard application programming interface (API).

2.3 ARINC 653 Application Partitioning

The ARINC 653 specification is one of the most important blocks from the Integrated Modular Avionics definition, where the partitioning concept emerges as a way to ensure protection and functional separation between applications. The partitioning concept aims enforcing fault containment, preventing fault propagation from one partition to another. It also eases system verification, validation and certification procedures.

The ARINC 653 standard specification describes a partition as being roughly the same as a program in a **single application environment** (comprising code and data, its configuration attributes and execution context) where one or more processes concurrently execute by sharing access to the processor infrastructure and other hardware resources.

The standard demands the functions resident on a core module to be partitioned with respect to space (memory partitioning) and time (temporal partitioning). A partition is therefore a program unit of the application designed to satisfy these partitioning constraints. A core module is responsible for enforcing partitioning and for the management of the individual partitions. Spatial (or memory) partitioning is achieved through allocation of predetermined areas of memory to each partition. Temporal partitioning ensures each partition uninterrupted access to common resources during predetermined time periods. Partitions are scheduled on a fixed, cyclic basis.

Configuration of all partitions throughout the whole system is expected to be under the control of the system integrator and maintained with configuration tables. The configuration table for the partition schedule will define the system major time frame (MTF).

2.4 ARINC 653 System Architecture

The general architecture of a standard ARINC 653 system is illustrated in Figure 1. It comprises the application software layer, with each application executing in a dedicated partition, and a given set of system partitions. The system partitions are optional components and are intended to manage the interactions with specific hardware devices. Appropriate support from the core software layer (e.g. hardware interfacing and device drivers) is required.

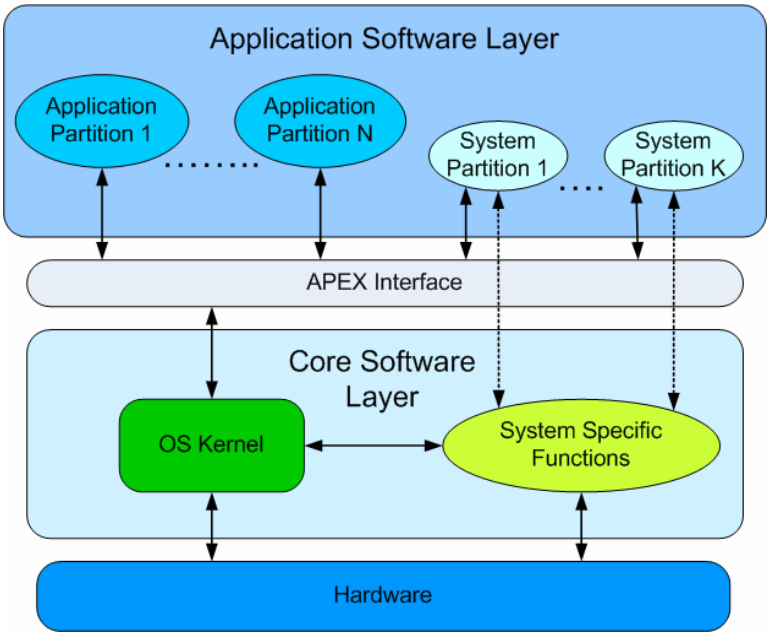


Figure 1 – Overview of the standard ARINC 653 System Architecture

Application partitions consist in general of one or more processes. The application partitions software exclusively uses the services provided by a **logical** Application Executive (APEX) interface, meaning it performs calls only to the set of primitives defined in the ARINC 653 standard specification. However, a system partition may use also a set of specific functions provided by the core software layer and may therefore bypass the standard APEX interface (cf. Figure 1).

In any case, the execution environment provided by the OS kernel module must furnish a relevant set of operating system services, such as process scheduling and management, time and clock management as well as inter-process synchronization and communication. The application software layer is obliged to strict robust space and time partitioning. Containment of possible faults inside the domain of each partition must be ensured by the core software layer.

Thus, each partition makes use of a **logical** execution environment. Given most of the process-level services that need to be supplied by the OS kernel module are already provided by common off-the-shelf real-time operating system (RTOS) kernels, a natural approach to the definition and design of ARINC 653 systems may use the functionality provided by the RTOS kernel, completed with the specific functions needed for ARINC 653 system operation, namely partition management mechanisms. From an architectural perspective the application executive (APEX) interface services are mapped into:

- the RTOS kernel canonical services, such as process management, time and clock management, inter-process synchronization and communication;
- the system specific OS and processor/platform functions, such as hardware interfacing and device driver functions, facilities for application downloading and support for built-in-self-testing;
- ARINC 653 specific functions, such as partition management and inter-partition communication.

The execution environment of each partition includes the corresponding application program (code and data), its configuration attributes and execution context.

2.5 AIR Definition Fundamental Issues

The key point in the definition and design of the AIR architecture do concern how the functionality provided by the RTOS kernel should be integrated with the ARINC 653 specific mechanisms that need to be added to the core software layer.

The additional components, introduced to support ARINC 653 specific mechanisms, must exhibit a well defined interface specification in order to secure the modularity and versatility requirements.

Furthermore, the RTOS native components needing to be modified for the integration in the AIR architecture should be restricted to a minimum.

3 AIR Architecture Design Specification

This section presents the AIR Architecture Specification, defining how the specific mechanisms required for conformity with the ARINC 653 standard specification can be effectively integrated with the functional structure of common RTOS kernels.

3.1 Defining the AIR Architecture

The fundamental architecture definition of the AIR system overall architecture is sketched in the diagram of Figure 2. It makes use of:

- a multi-executive core software layer;
- a two-level hierarchical scheduler.

A different instance of a RTOS kernel is used per partition. Even if the RTOS integration is homogeneous, i.e. even if the same RTOS kernel (e.g. RTEMS) is being used in every partition, these should be regarded as different RTOS kernels, with its own process scheduler. This structure allows the implementation of the hierarchical scheduler design approach.

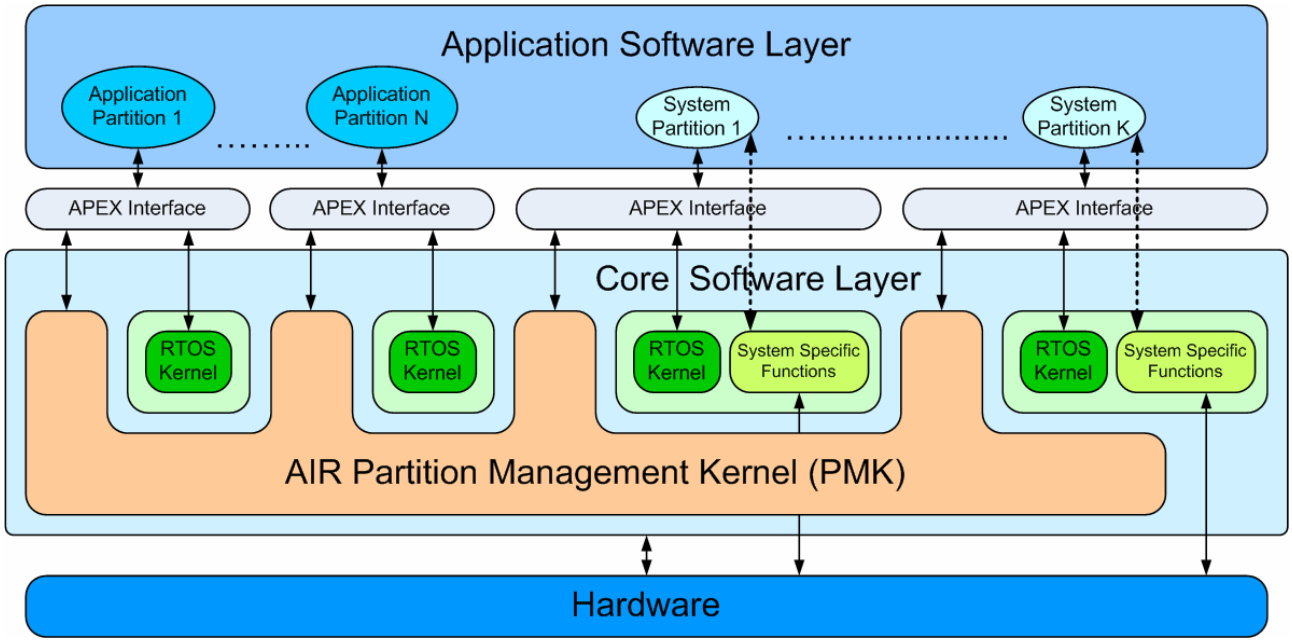


Figure 2 – Overview of the AIR System Architecture

The integration of the different components in the architecture of Figure 2 assumes a well-defined specification of their interfaces in order to preserve the modularity characteristics of the architecture. The components added to the native RTOS kernel aim to supply the functionality missing in those operating systems, to be in conformity with the standard ARINC 653 specification. The fundamental AIR design components integrated in the architecture of Figure 2 are the following:

- the **application executive (APEX) interface** defining, for each partition in the system, the set of services defined in the ARINC 653 standard specification;
- the **native RTOS kernel**, which in conformity with the architectural attributes of the multi-executive core design can be specifically configured for each partition;

- the system partitions do an additional use of **system specific OS functions**, in general supplied also with the native RTOS distribution and that, again, may be individually configured in a per-partition basis;
- the **AIR Partition Management Kernel (PMK)**, made from a fundamental set of functional modules providing the specific functionality that needs to be added for conformity with the ARINC 653 standard specification.

The **application executive (APEX) interface** should be designed as much as possible by mapping the ARINC 653 service primitives into the service interface of the native RTOS kernel, i.e. into the native RTEMS primitives [RTEMSAPI], for the particular case of the current AIR architecture. As a design alternative, other OS interfaces may be used, such as the POSIX interface, also offered by RTEMS [RTEMSPOSIX]. In some specific cases, where the calling conventions or the functionality provided does not exactly match, a more complex design approach may be required.

In addition, the APEX interface may need to use specific services provided by the AIR Partition Management Kernel (PMK), allowing the invocation of specific functions and the exchange of system specific information such as partition configuration tables, partition status and operating modes, as well as other system-level parameters.

The use of highly configurable **RTOS kernels** allows an effective adaptation of the services provided by the RTOS kernel to the requirements of each partition. This is especially true for system partitions, where in addition to the standard application executive (APEX) interface there is a need for system specific OS functions.

The **system specific OS functions** in essence establish a set of interface stubs to address the specific details of each hardware platform. In conformity with the ARINC 653 standard specification, this functionality is only allowed in system partitions. However, even for system partitions, special attention must be given by the system integrator so that two partitions do not collide when controlling the same hardware. Usually the hardware is accessed through device drivers, so two system partitions should not access the same driver unless the driver itself supports this requirement.

Finally, the **AIR Partition Management Kernel (PMK)** includes the fundamental modules required to support the standard ARINC 653 functionality complemented with a set of complementary modules required to make the integration of the RTOS kernel and of the corresponding specific system OS functions effective. The AIR Partition Management Kernel (PMK) also has an attachment for the application executive (APEX) interface, intended for the provision of the AIR inter-partition communication services and other relevant system functions.

3.2 AIR PMK – The AIR Partition Management Kernel

The main components of the AIR Partition Management Kernel (PMK) include the following functional modules:

- **AIR inter-partition communication**, allowing the exchange of information between different partitions without violating spatial segregation constraints.
- **AIR partition dispatcher**, which has the responsibility of saving the execution context of the partition to be deactivated (the currently running partition) and of restoring the execution context for the partition to be executed next (the heir partition). It also secures the management of spatial segregation.
- **AIR partition scheduler**, selecting at given times which partition owns system resources, namely the processing infrastructure. In conformity with the ARINC 653 standard specification, a cyclic fixed scheduler is used.

- **AIR time manager**, which has the responsibility of processing the strict time references provided by the hardware system clock, forwarding them to the appropriate system components in the AIR architecture.
- **AIR Hardware Abstraction Layer (HAL)**, providing an effective level of abstraction for accessing the hardware resources.

The **AIR inter-partition communication** module is responsible for the encapsulation and transference of data from one partition to another in a spatial segregated environment. This implies the management of the corresponding memory protection mechanisms.

The **AIR partition dispatcher** is responsible for switching between partitions. It starts by saving the execution context of the partition being deactivated. Activating the next partition implies restoring the execution context for the partition, including the management of the spatial segregation mechanisms.

The **AIR partition scheduler** is responsible for securing time segregation, by restricting the processing time assigned to each partition, as defined by ARINC 653 system configuration. The ARINC 653 partition scheduler assigns the processor infrastructure to each partition in a cyclic fashion, within the period of a major time frame defined by system configuration. In the context of the AIR System Architecture, using a **two-level hierarchical scheduler** approach, the native RTOS kernel (e.g. RTEMS) is responsible for the scheduling of the processes inside each partition.

The **AIR time manager** should keep a record of the system clock ticks for latter announcement to the RTOS process scheduler of inactive partitions. This deferred announcement of system clock ticks is performed upon partition switching. The running partition and the corresponding RTOS process scheduler are immediately notified of each clock tick.

A specific **AIR Hardware Abstraction Layer (HAL)** should be included in the AIR Partition Management Kernel (PMK) for the processor architecture, processor family variant and hardware platform supported by a given AIR system. Within the scope of AIR project two case studies are currently addressed: the SPARC ERC32 and LEON processor cores; the Intel IA-32 (80x386) architecture.

The mapping of system specific OS functions related to the management of spatial segregation into the specific details that characterize a given processor architecture could be effectively handled by the AIR Hardware Abstraction Layer (HAL).

Other fundamental issues which have a significant impact in the design of the AIR Hardware Abstraction Layer (HAL) concerns the management of input/output (I/O) related functions and the management of interrupts and therefore of interrupt handlers. For example, the AIR Hardware Abstraction Layer (HAL) can provide a set of primitives that allow the establishment of interrupt handlers by the system partitions. The management of interrupts and of interrupt handlers is critical in the AIR Architecture and should be performed on a per-partition basis. This avoids that external events triggered inside the time window of a given partition may cause temporal interference in the running partition. Avoiding interrupt handlers associated to a given partition to be executed on the context of other partition also helps to preserve the integrity of the system.

The structure and the design of the AIR Partition Management Kernel (PMK) components, such as the AIR partition scheduler and dispatcher, should be made as much as possible independent of a given operating system. This contributes to a modular and flexible design of those components and therefore for a highly modular AIR Partition Management Kernel (PMK). All the processing infrastructure dependencies shall be included in AIR Hardware Abstraction Layer (HAL).

The AIR Partition Management Kernel (PMK) does not exhibit strict configuration requirements. The presence of almost all AIR Partition Management Kernel (PMK) functional components is mandatory. The only exception concerns the AIR inter-partition communication module, which only needs to be included in the architecture when such services are required.

The architecture of the AIR Partition Management Kernel (PMK) also needs a specific component to be included:

- **AIR partition manager**, which has the responsibility of performing the general initialization of the system, including invocation of all specific RTOS initialization managers. The AIR partition manager also allows reporting at the APEX interface specific AIR Partition Management Kernel (PMK) data, such as partition configuration tables and other system-level parameters.

The **AIR partition manager** is a fundamental component of the AIR PMK architecture in the sense it is responsible for the general initialization of all AIR PMK components including the initialization of each specific RTOS kernel. This concerns the preparation for starting multi-tasking mode and process execution in the RTOS kernel of each partition. Setting up multi-tasking mode and therefore process execution can only be started in one (and only one) partition after all partitions have been prepared to enter multi-tasking mode. This has been revealed as one key point in the design of the AIR PMK Multi-Executive Core architecture.

3.3 AIR Process Scheduling Hierarchical Integration

The AIR architecture uses a two-level scheduler hierarchical approach to support the integration and effective interaction between the AIR Partition Scheduler and the scheduler provided at each partition for the scheduling of the processes contained in each partition.

The hierarchical integration of RTOS kernel schedulers in a multi-executive environment assumes that a different instance or even a completely different RTOS kernel is used for each partition. Each partition has its own instance of a RTOS process scheduler.

A single **flat scheduler** performs partition scheduling and selects at given points in time which will be the next running partition. This is the function of the AIR PMK Partition Scheduler, which performs a periodic fixed cycle scheduling.

A **RTOS process scheduler** is used for process scheduling inside each partition. A given RTOS process scheduler becomes active upon the switching to the given partition. All the process scheduling operations, including clock tick processing, is restricted to the scope of the active partition. Thus, the temporal interference between partitions is minimal with this regard.

All the overheads concerning clock tick processing and process status update do arise upon partition context switching. This delay is a function of the number of clock ticks that need to be processed and of the number of processes that require a status update. However, for most of the commercial off-the-shelf RTOS kernels this operation can be optimized. At most, it will exhibit only a slight penalization of the system overall timeliness values.

The hierarchical integration of process schedulers simplifies the utilization of different RTOS kernels in each partition. No fundamental changes are needed in the RTOS native scheduler. It also allows the utilization of diversified set of process schedulers.

In conformity with the ARINC 653 standard specification, partitions use a priority-based preemptive process scheduler. However, the AIR architecture two-level hierarchical process scheduler makes simple the integration of other schedulers in a per-partition basis. Some partition may use a different kind of scheduler, such as rate monotonic, deadline monotonic, earliest deadline first or least laxity first.

Most of the commercial off-the-shelf RTOS kernels perform process scheduling according to a preemptive priority-based algorithm. The RTOS process scheduler selects for the running state the highest priority ready process, i.e. the highest priority process that has all the resources required for execution with the possible exception of the processor.

3.4 AIR RTOS Integration

The integration of a given RTOS in the AIR Architecture should follow as much as possible a very modular strategy. All the specific ARINC 653 functionality should be supported by AIR specific components integrated in the design of the AIR Partition Management Kernel (PMK).

The addition (or in some cases the migration from the RTOS distribution) of the AIR Hardware Abstraction Layer (HAL) and all the corresponding functionality (management of hardware resources, such as system clock, interrupt controllers, input/output ports) is a step towards a modular integration of the RTOS Kernel.

On the other hand, only the set of services effectively required by the APEX interface needs to be included in the RTOS instance for that partition. For example, all the services related with dynamic memory allocation are usually excluded from ARINC 653 compliant designs in order to ensure bounded processing times. Thus, those services do not need to be included in any instance of the RTOS Kernel. This is true for both application and system partitions. This property is closely related with the configurability characteristics of each RTOS Kernel.

3.4.1 AIR RTOS Integration: RTEMS

Conforming to the AIR architecture, RTEMS, the Real-Time Executive for Multiprocessor Systems offers interesting characteristics to be integrated in its design. One aspect concerns the interest around the RTEMS Kernel, given its qualification for space on-board software developments [RTEMSQUAL]. On other hand, it is able to provide the functionality required for the execution of each individual partition, in the sense it provides a comprehensive set of services such as preemptive process scheduling, process management, time management and inter-process synchronization and communication. This functionality is required not only for the fundamental temporal requirements of the ARINC 653 specification[1] but also for the provisioning of an extensive set of primitives that allow a mapping with the APEX interface.

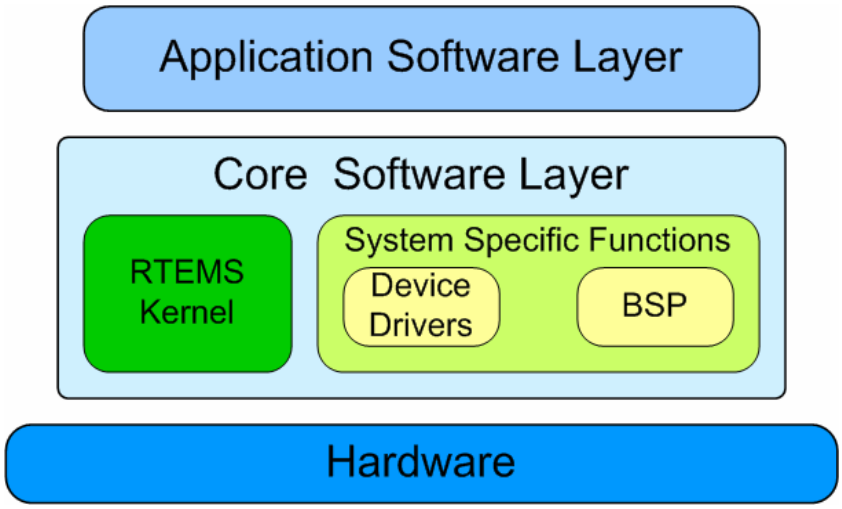


Figure 3 – Typical structure of a RTEMS Application

The RTEMS architecture is illustrated in Figure 3, where the resemblance with the AIR architecture defined in Figure 2 is noticeable. The RTEMS kernel executes in a diversified set of processor platforms. Included in the system specific OS functions, the RTEMS BSP (Board Support Package) module contains the methods to address a specific hardware target, leaving the RTEMS kernel unmodified throughout several processor

[1] Processes scheduled by a preemptive priority-based algorithm.

families. In fact, the engineering of a new RTEMS BSP is not hard to achieve. Additionally, the RTEMS device driver module introduces the methods by which the hardware devices are accessed.

Furthermore, each RTEMS intrinsic component only needs to be included in a given application and/or system partition when required. For example, to preserve timeliness none of the RTEMS dynamic memory related modules (e.g. region and dual-ported managers) is included in the AIR architecture.

3.5 Building the AIR Architecture and its Components

The profile represented in Figure 4 illustrates some of the engineering issues concerning the usage of canonical development tools in the production of the AIR system. For each component in the system, i.e. the AIR Partition Management Kernel (PMK), the RTOS Kernel, the APEX Interface and the application using it at each partition, the illustration in Figure 4 identifies the corresponding program (code and data), its configuration data and execution context (stack) memory spaces.

Entirely canonical build tools (e.g. cross compilers, linkers, libraries and library managers and other object level tools) may be used inside each partition. However, the use of the same RTOS Kernel in more of one partition in the scope of the multi-executive core approach implies a non-standard use of the RTOS production chain in order to cope with a common RTOS Kernel naming structure at each partition.

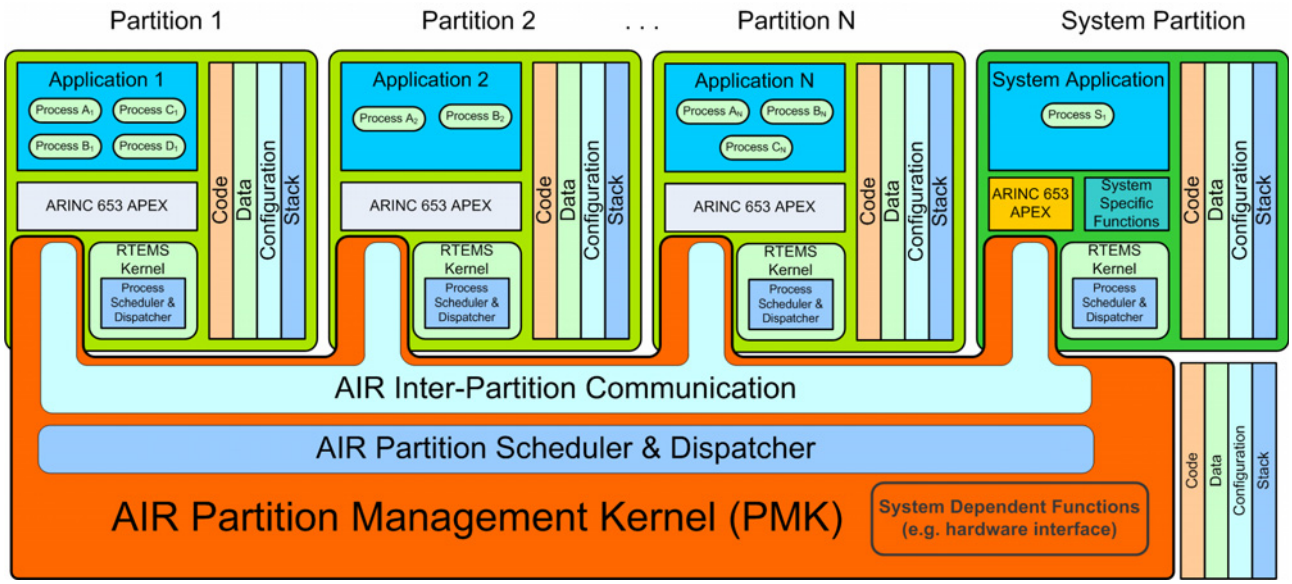


Figure 4 – Building an AIR System

3.6 Summary of AIR Architecture Attributes

The AIR architecture is built from a **multi-executive core two-level hierarchical multi-scheduler solution**. The engineering of this solution requires an effective integration of AIR components with the RTOS object libraries and with the corresponding application production chain. A limited number of RTOS components require effective integration at the source level.

The architectural attributes of the multi-executive core design in respect to the definition of a service interface conformant with the ARINC 653 standard, using a commercial off-the-shelf RTOS kernel such as RTEMS, is summarized in Table 4.

Multi-Executive Core Architecture Attribute								
RTOS Integration	**Versatility**	**Modularity**	**Configurability**	**Integrity**	**Fault Confinement**	**Footprint Size**	**Bootstrap Method**	**Development Tools**
partitioned	good	good	good	good	good	non optimal	single-file	canonical plus object filtering
							multi-file	canonical plus bootstrap tool

Table 4 – Summary of AIR Multi-Executive Core Architecture Attributes

The multi-executive core design allows a high integrity partitioned environment for the execution of applications which can be made highly effective in respect to: versatility, modularity, configurability and confinement of faults to a partition domain. Since multiple instances of the same RTOS are in general present, the corresponding footprint size is non-optimal.

A multi-executive core solution allows two methods for application deployment. The single-file approach is in strict conformity with canonical bootstrap methodologies, such as those provided by the GRUB tool, for both real and synthetic processor targets. However, it requires at least an object file processing step before the final linking into a single application file may take place.

The multi-file approach allows applications to be built into different application files using a completely canonical methodology. Unfortunately, it requires a non-canonical method for system bootstrapping.

Process Scheduler Integration Attribute							ARINC 653 Core Target
Partition Scheduler	**Process Scheduler**	**RTOS Kernel**	**Versatility**	**Modularity**	**Partition**		
					Switching Overhead	**Temporal Interference**	
flat	hierarchical	homogeneous	good	good	slight	minimal	**multi-executive**
		heterogeneous					

Table 5 – Summary of AIR Process Scheduler Integration Attributes

The attributes of the two-level hierarchical integration of partition and process schedulers in the partitioned multi-executive core architecture are summarized in Table 5.

A single first-hierarchy flat scheduler is used to select which will be the running partition according to a fixed cycle schedule. Inside each partition, processes are scheduled in a second-hierarchy level using the native process scheduler of the RTOS deployed in that partition. Both homogeneous (same RTOS in all partitions) and heterogeneous (a different RTOS in some partition) solutions are possible, exhibiting in both cases almost optimal properties in respect to timeliness attributes.

4 AIR Mechanisms for Spatial and Temporal Segregation

This section thoroughly describes the specific mechanisms defined in the AIR architecture to secure segregation of applications in both spatial and temporal domains.

4.1 Spatial Segregation

The ARINC 653 standard imposes that the software components operating in a given partition, exception made for inter-partition communication mechanisms, cannot access the addressing space of other partitions (or at the least cannot do so with write privileges). This mechanism is known as spatial segregation and it implies the use of protection mechanisms in the access to a given addressing space. This may involve protection of both memory and input/output (I/O) addressing spaces. Partitioning of the addressing space prevents applications of interfering with each other.

To be effective these mechanisms should be supported directly in hardware address validation mechanisms, typically implemented in Memory Management Units (MMU). In essence two kind of memory protection for an addressing space do exist.

The first method simply performs the monitoring of the addressing spaces and the detection of address access violations. Given its simplicity a hardware unit performing only the monitoring of addressing spaces is sometimes simply referred as a Memory Protection Unit (MPU). The diagram in Figure 5 illustrates the fundamental mechanisms for the monitoring of addressing spaces and protection through the detection of address access (range and/or mode) violations.

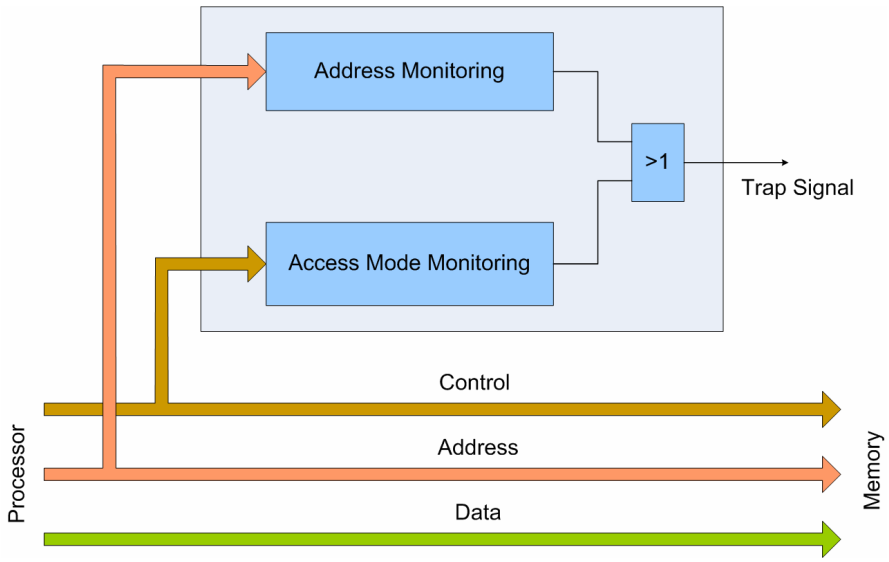


Figure 5 – Memory Protection through Monitoring of Addressing Spaces

The diagram in Figure 5 specifies a continuous monitoring of the CPU address bus: the contents of the CPU address bus are compared with the valid address range, either by specifying the beginning and the ending of the allowed addressing space or through the specification of base address and address space size parameters. The two methods are roughly equivalent and do exhibit a similar complexity.

In addition to the monitoring of the addressing range, the memory access mode can also be monitored. For example, a given address space can be configured for: read access only; read and write access; read and execute; execute only; etc.

A basic memory protection scheme, employing the monitoring of the RAM addressing space, is used in the SPARC LEON2 implementation of the ATMEL AT697E memory write protection mechanisms.

Memory protection methods based on address monitoring are simple to implement and, as referred in [RUSHB1], are methods inherently less vulnerable to bit-flips caused by single-event upsets (SEU).

Furthermore, such methods can provide in general a fine-grained protection of different kinds of memory addressing spaces. In the context of the AIR architecture, memory protection based in address monitoring exhibits the following advantages:

- simplicity of implementation and management;
- possible combination with access modes classification and other protection mechanisms;
- possible fine-grained range of address protection (e.g. double word natural boundary in 32-bit processors);
- an inherent low susceptibility to bit-flips caused by single-event upsets.

On the other hand, memory protection based address monitoring methods may be too simple and therefore slightly inflexible in the support of AIR Inter-partition Communication schemes, in the sense the system designer/integrator may wish to provide different mappings for the transmit and receive ports of a given communication channel, for example.

The second method performs a translation of the valid addressing space: a given logical address is translated into a (linear) physical address, usually from a base address specified through a Memory Management Unit (MMU) descriptor. This method exhibits a complexity similar to the detection method. The address translation option provides additional flexibility in respect to the mapping of a logical address space into a physical address space. In addition, it also allows the mapping of different logical addresses in the same physical address references thus creating some sort of address alias. However, it potentially exhibits a higher vulnerability to bit-flips caused by single-event upsets.

A fault in the memory translation mechanisms should be regarded as very serious with regard to the implementation of the memory protection mechanisms required by the ARINC 653 standard specification. Given address translation may be a useful mechanism for the implementation of AIR inter-partition communication, one possible solution to increase fault coverage would be the combination of memory translation and memory detection schemes in this particular case.

The applications run in non-privileged mode and each memory access is checked (and sometimes translated) by the processor Memory Management Unit (MMU), using memory and/or I/O addressing descriptors tables. A trap signal is generated upon the detection of an unauthorized access to an addressing segment. The memory translation scheme is used in the segmentation model of the Intel IA-32 architecture.

The design of these mechanisms and its use within the scope of the ARINC 653 standard should either incorporate **fault-tolerant mechanisms** at design level or use a **combination** of memory translation and memory detection mechanisms. In addition, they should also prevent ill interactions with other advanced addressing resources such as processor internal data and instruction caches.

Other subtle processor architecture details, such as hidden MMU descriptor registers should also be taken into account in the design and/or utilization of the mechanisms enforcing spatial segregation. Furthermore, an adequate combination of hardware address checking mechanisms with software-based approaches, such as Software Fault Isolation (SFI), may provide additional dependability coverage guarantees.

The provision of spatial segregation mechanisms exhibiting a high fault coverage and conformity with the ARINC 653 standard specification are foreseen to be integrated in the operation of the AIR Partition Management Kernel (PMK). The management of such mechanisms should be included in the design of the AIR Partition Dispatcher with the implementation of some functions delegated to the low-level AIR Hardware Abstraction Layer (HAL).

In particular, these AIR Partition Management Kernel (PMK) modules need to provide memory and/or I/O addressing space management for the following specific functions:

- handling of hardware interrupt sources, namely the system clock tick interrupt;
- operation of AIR PMK components (Partition Scheduling and Dispatching) upon system clock tick notification;
- normal operation of a given AIR application partition;
- normal operation of the corresponding APEX interface;
- normal operation of the APEX – AIR PMK interface;
- management of AIR inter-partition communication.

The functionality required for the management of such memory and/or I/O addressing spaces can be mapped into:

- a segment-based model, implementing either an address monitoring or an address translation scheme, or a combination of both;
- a page-based model, which usually uses address translation schemes.

The Intel IA-32 architecture uses both the segmentation and page-based models while the specification of the SPARC V8 architecture (ERC32, LEON2, LEON3) uses a three-level paging scheme. Both the approaches can be used for the mapping of the different AIR functional elements in the memory protection resources. For example, the AIR inter-partition communication module may use either a segmentation model or a paging-scheme to create alias in the different partitions for the same communication channel.

Within the scope of the ARINC 653 standard specification there are no provisions to spatial segregation at process-level inside each partition.

4.1.1 Memory Protection Mechanisms in SPARC LEON Cores

Though the bare SPARC V8 architecture does not imposes the implementation of a given MMU, some common processor core designs may include either some simple memory protection schemes or more complex units using a memory paging mechanism compliant with the SPARC V8 specification. We address next both situations.

ATMEL AT697E SPARC V8 LEON2-FT Processor

This implementation of the SPARC LEON2 architecture has a 32-bit address bus that allows the addressing of external address spaces, in function of a given memory type. With respect to external RAM memory space, the corresponding attributes are summarized in Table 6.

Memory Type	Address Range	Size	Write Protection
RAM	0x40000000 – 0x7FFFFFFF	1 GiB	✓ (2 units)

Table 6 – Characteristics of ATMEL AT697E RAM Address Space

The ATMEL AT697E processor allows protection of the external RAM address space through two write protection units. These write protection units[2] are simple segment-oriented mechanisms for external RAM

[2] A simple form of this mechanism is usually referred in the literature as a *fence register*.

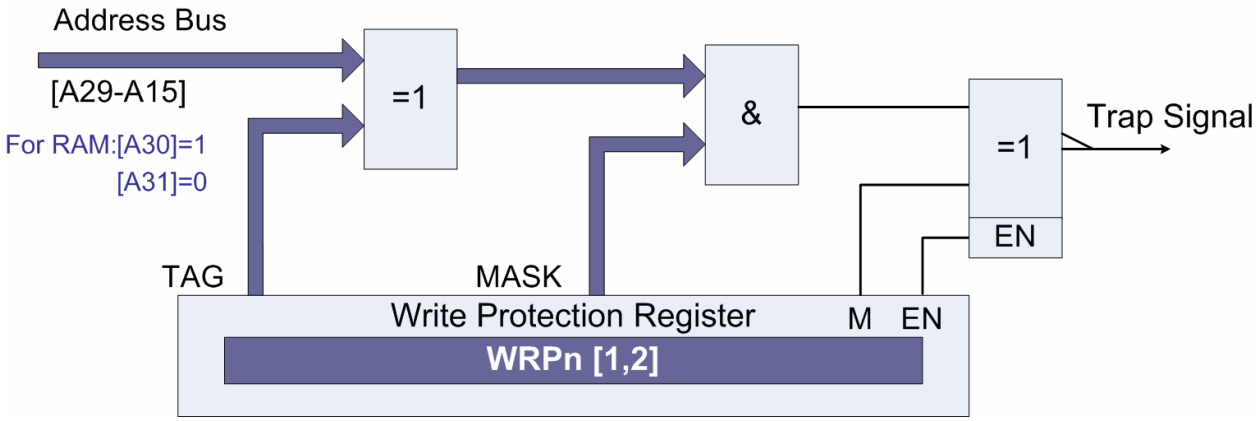
write protection to be performed over 32 KiB blocks[3]. Sets of such write protected memory segments can be located in contiguous or interleaved memory spaces.

The RAM write protection functions are configured and controlled through two write protection registers. Their functional details are summarized in Figure 6. The mechanism specifies the following fields:

- a **TAG address**, which defines fifteen of the most significant bits [A29-A15] of the address block/segment to be protected;
- a **MASK,** specifying which bits of the tag address should be monitored. This includes:
  - a (possibly empty) set of zero bits at the least significant positions, defining the size of the memory block, which must be always a power of two multiple of 32KiB. If no such set exists, the size of the memory block is 32 KiB.
  - a (possibly empty) set of zero bits at other positions but the least significant ones, defining a group of interleaved memory blocks. If no such set exists, a single (non-interleaved) memory block is defined in the address space.
- two **protection modes** can be specified: active within the segment or active outside the segment.
  - M=0 – the exterior of the segment is write protected (segment mode)
  - M=1 – write protection within the segment (block mode)
- **enable/disable** of the write protection features:
  - EN=0 – no write protection
  - EN=1 – write protection is active (enabled)

When the RAM write protection mode is active, if an address range violation is detected the memory write cycle is stopped and a trap (0x2B) error is generated.

This RAM write protection function roughly corresponds to the minimum mechanisms required to support the AIR (and therefore ARINC 653) spatial segregation requirements, as illustrated in Figure 7.



WRP bit	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	1	3	2	1	0
Field	**EN**	**M**	**TAG Address**															**MASK**															
Meaning	control		fifteen of the most significant address bits															32 KiB protected block															

Figure 6 – The ATMEL AT697E SPARC LEON2 RAM Memory Protection Mechanisms

[3] This document follows a notation in conformity with the IEC 60027-2 standard in respect to the usage of prefixes for the multiples of binary quantities.

For example, one of the ATMEL AT697E write protection registers may be used for the protection of the AIR Partition Management Kernel (PMK) address space (cf. Figure 7). This needs to include:

- handling of hardware interrupt sources, namely the system clock tick interrupt;
- normal operation of the AIR APEX – PMK interface, through the AIR PMK Partition Management module;
- the management of AIR inter-partition communication channels.

The second write protection register may be used to enable a given running partition addressing space (cf. Figure 7). This includes:

- the application code, data and stack spaces;
- the RTOS kernel instance for the partition;
- the corresponding AIR APEX interface components.

Enabling a valid addressing space for the running partition has to be managed by the AIR PMK Partition Dispatcher and by the AIR Hardware Abstraction Layer (HAL), upon partition switching. This may also need to include the management of AIR inter-partition communications.

The mapping of partition (and of AIR PMK) addressing spaces in segments with a size multiple of 32 KiB needs also to be managed at the application production chain level, e.g. using the linker/locater address alignment features.

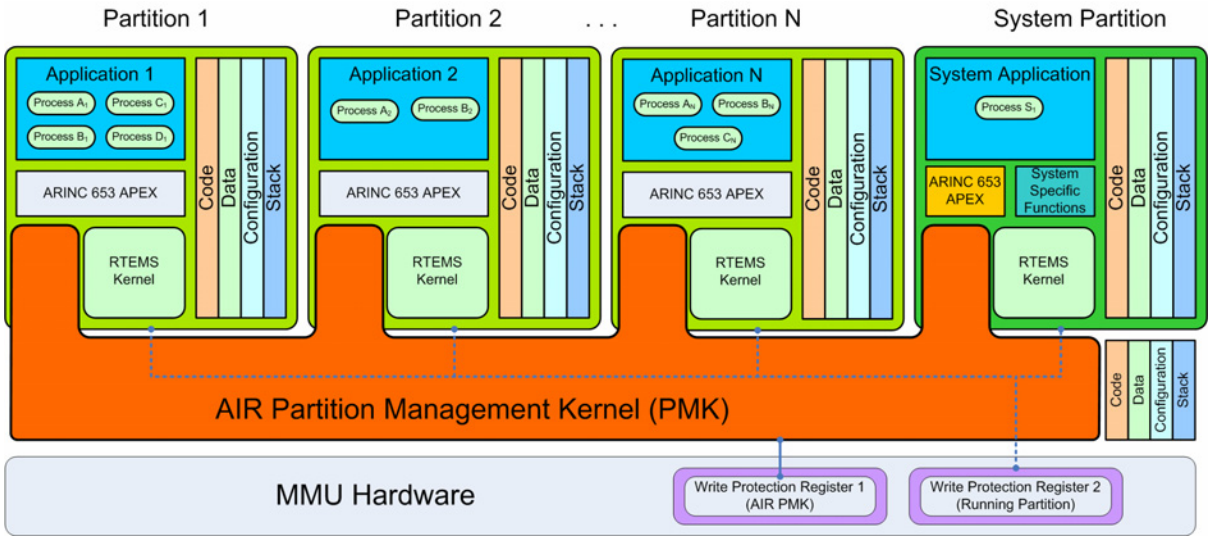


Figure 7 – Usage of ATMEL AT697E RAM Memory Protection Mechanisms in AIR

Gaisler LEON2-FT and LEON3-FT SPARC V8 Processor Cores

These implementations of the SPARC LEON3 architecture includes an optional component that implements the Memory Management Unit (MMU) described in the SPARC V8 specification. In essence this document specifies a three-level paging scheme for translating a 32-bit virtual page address into 36-bit physical page address. The page size is 4 KiB. The operation of this mechanism is illustrated in Figure 8.

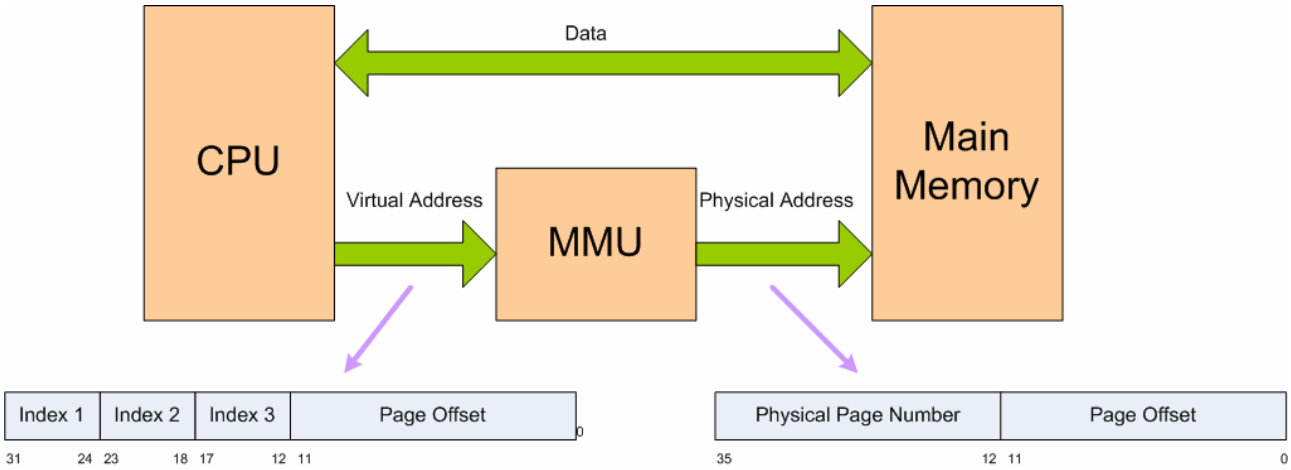


Figure 8 – The Gaisler SPARC LEON3 SPARC V8 MMU Architecture

On the other hand, the operation of the Gaisler SPARC LEON3 MMU is illustrated in Figure 9, showing the full address translation (i.e. from virtual to physical) scheme. The mapping of AIR system protection requirements can then be summarized as follows:

- context level (root pointer) – specifies the active partition reference;
- partition level (index 1) – also specifies the active partition;
- entity level (index 2) – specifies the AIR component in use (application, APEX interface, RTOS,…).
- object level (index 3) – specifies one particular element in the AIR component (e.g. a channel in the AIR inter-partition communication module).
- method (page offset) – specifies the specific function or functional element being accessed.

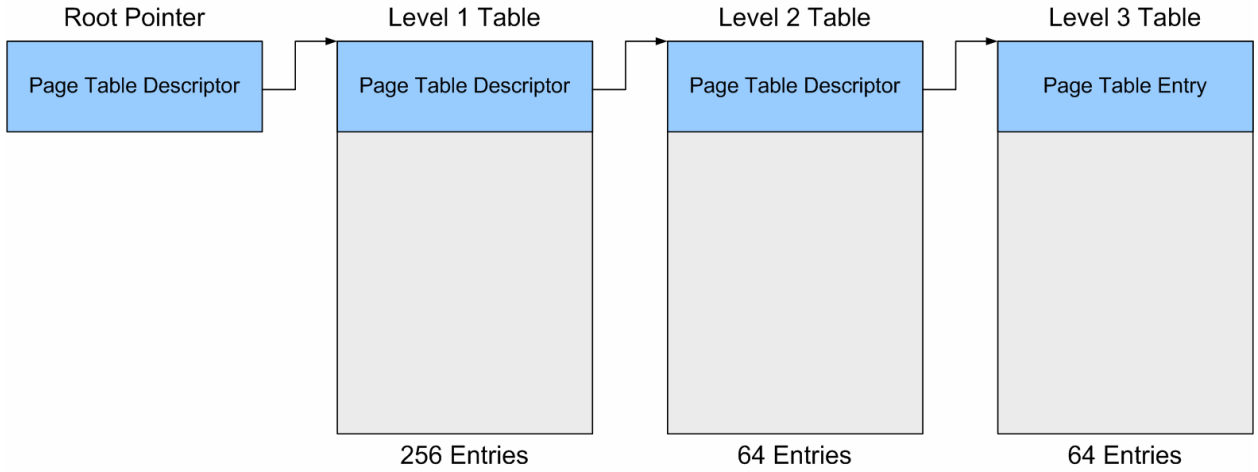


Figure 9 – The Gaisler SPARC LEON3 MMU Operation

4.1.2 Memory Protection Mechanisms in Intel IA-32 (80X86) Processors

The Intel IA-32 processor architecture directly provides hardware-based segment mediation allowing address translation (and validation), through the use of the so-called segmentation mechanisms. The fundamental operation of this mechanism is illustrated in Figure 10.

Using, the IA-32 nomenclature a segment is a protected addressing space, which can be used to support ARINC 653 partitions. The memory management mechanisms provided by the IA-32 architecture are in essence a memory address translation and validation scheme. An exception (trap) is generated upon the detection of an unauthorized access to a given addressing segment.

The Intel IA-32 address translation and validation mechanism works as follows. The processor logical address can be divided in two components:

- a segment selector, usually stored in one of the segment registers (for example: by default, the CS register is used for code references, the DS register is used for data and the SS register is used while accessing the stack)
- an offset, also known as effective address, specifies the relative displacement of a given address reference with respect to the base address of the segment.

The base address for the segment (i.e. the physical address where the segment begins) and its limit (i.e. the size of the segment) are specified in a descriptor located in a memory space under the supervision of the memory management unit (MMU). Under normal operation (i.e. if the offset does not exceed the specified limit) the value of the offset is added to the base address to generate the linear physical address. If the offset value exceeds the segment limit an exception (trap) is generated and the memory access is not performed.

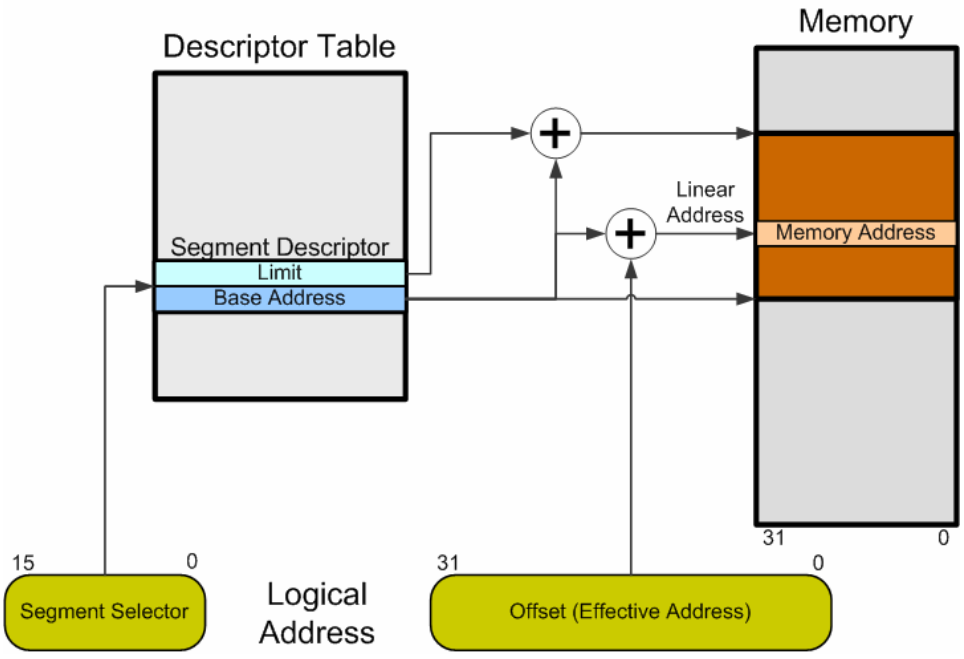


Figure 10 – The Intel IA-32 Segment-based Memory Protection Mechanisms

No particular memory alignment requirements are placed either on the AIR PMK components or on the AIR Application modules: all these components can be aligned on the Intel 80x86 32-bit natural boundaries. The usage of physical memory can be very effective. In addition, no particular limitation is placed on the usage of canonical development tools with this regard.

As an alternative the Intel IA-32 architecture also offers the possibility of using a two-level paging scheme, as illustrated in Figure 11. These mechanisms, thought simpler, are similar to those used in the Gaisler SPARC LEON3 architecture.

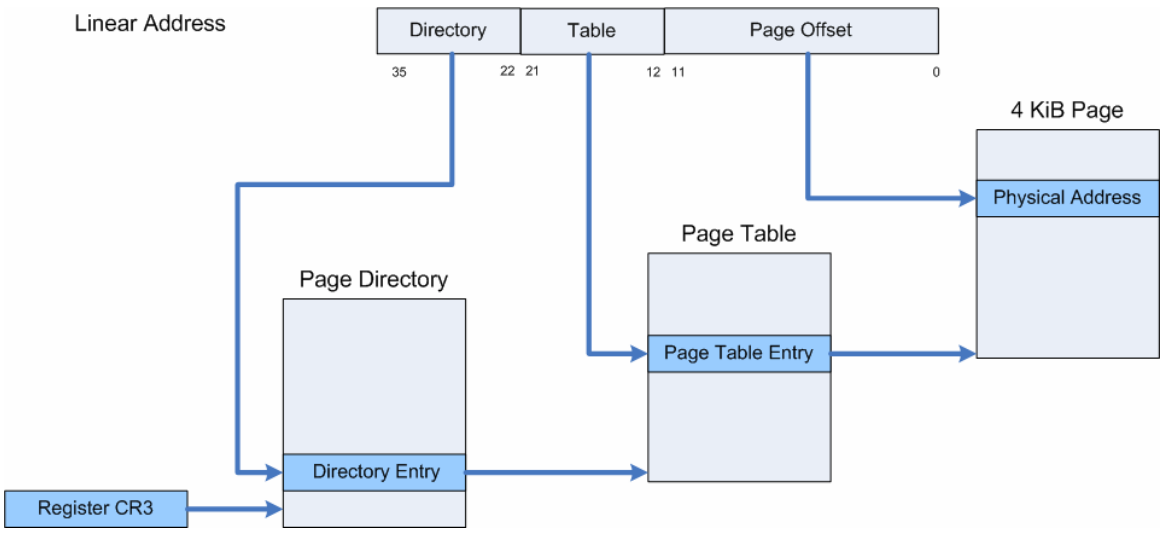


Figure 11 – The Intel IA-32 Page-based Memory Protection Mechanisms

4.2 Temporal Segregation

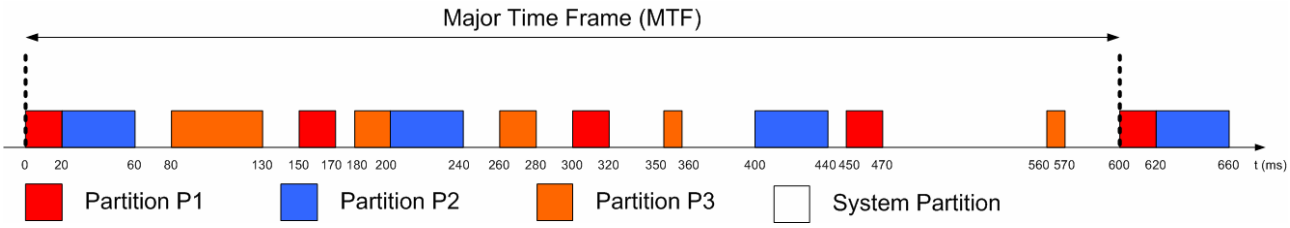
Temporal segregation in the ARINC 653 standard specification enforces strict time windows for each partition. This ensures that each partition has a given execution period, even if other partitions are faulty. Partitions are scheduled on a fixed, cyclic basis. Inside the time windows of a given partition, its processes are scheduled preemptively according to their priority and state.

The AIR partition scheduler determines at each point in time what should be the running partition, based on a time table defined off-line. It determines, at partition preemption points, the next running partition: the heir partition. The AIR partition dispatcher is responsible for replacing the running partition with the heir partition.

The AIR architecture follows a multi-executive two-level hierarchical multi-scheduler approach. This means that each partition provides its own process scheduler, usually through the native priority-based preemptive scheduler of each RTOS kernel.

4.2.1 Partition Scheduling

The scheduling of partitions defined by the ARINC 653 standard specification is strictly deterministic over time, that is to say, each partition has a fixed temporal window in which it has control over the computational platform. Furthermore, each partition is scheduled on a fixed, cyclic basis. The Partition Management Kernel maintains a Major Time Frame (MTF) of fixed duration, defined off-line, which is periodically repeated throughout the runtime operation [A653P1]. An example of the temporal partitioning is provided by Figure 12.

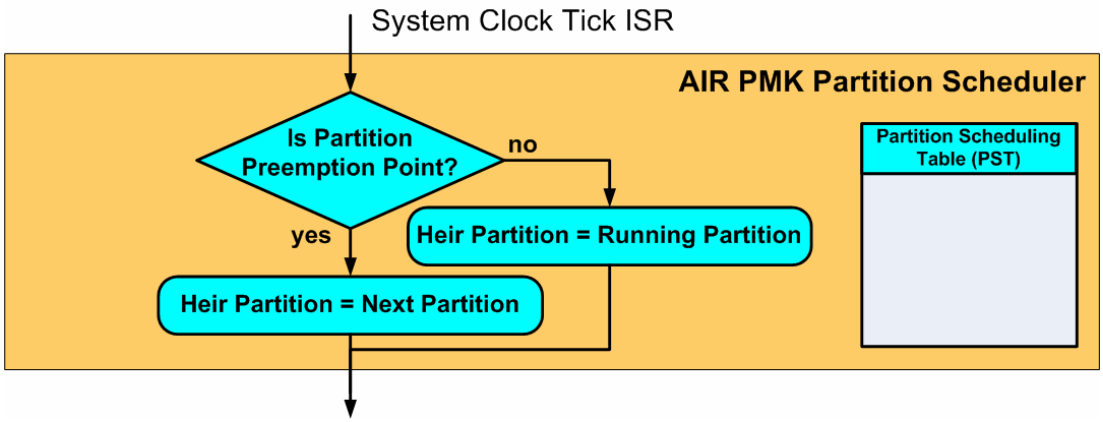


Partition Scheduling Table			
Clock Ticks	**Heir Partition**	**Clock Ticks**	**Heir Partition**
0	P1	280	System
20	P2	300	P1
60	System	320	System
80	P3	350	P3
130	System	360	System
150	P1	400	P2
170	System	440	System
180	P3	450	P1
200	P2	470	System
240	System	560	P3
260	P3	570	System

Figure 12 - Example of Partition Scheduling over a Major Time Frame

The AIR Partition Scheduler is a component of the AIR Partition Management Kernel (PMK) that needs to be integrated with the RTOS kernel according to the multi-executive two-level hierarchical multi-scheduler architecture. .

The Partition Scheduler only needs to be activated to take a decision on the next running partition at specific points in time. These points are defined by the Greatest Common Divisor (GCD) of the partitions preemption points. Let us define the GCD period as a multiple of the system clock tick. Thus, it is only required that the partition scheduling be executed at each clock tick. The AIR PMK Partition Scheduler requisites and functionality is summarized in Figure 13.



AIR PMK Partition Scheduler	
Pre-requisite	
	Activated at clock ticks
Function	
	Determine whether the current clock tick is a partition preemption point
	If so, determine the heir partition

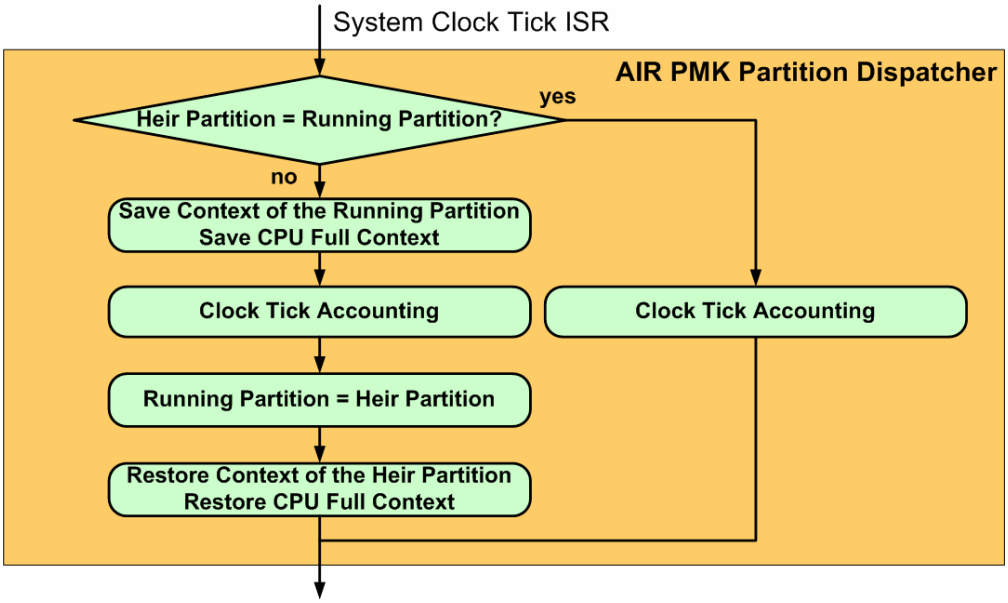
Figure 13 - AIR PMK Partition Scheduler functionality

The AIR PMK Partition Scheduler is responsible for selecting the next running partition (the heir partition) at each system clock tick, along the Master Time Frame (MTF) period. The AIR PMK Partition Scheduler pre-requisite can be fulfilled in general by common computational platforms, such as those based in the SPARC ERC32/LEON processors and the Intel IA-32 architecture, through its integration in the system clock interrupt service routine (ISR). By its integration in the system clock interrupt service routine, the partition scheduler has only to determine if the clock tick that triggered the interruption is a partition preemption point and if so, determine the heir partition and enable partition dispatching.

The partition scheduling itself can be computed very simply, based on the number of elapsed clock ticks and on a table (the Partition Scheduling Table, PST) containing the preemption clock ticks and the corresponding heir partition. The Gantt diagram of a given partition scheduling and the contents of the PST for the given example, are illustrated in Figure 12.

4.2.2 Partition Dispatching

After partition scheduling, the AIR PMK Partition Dispatcher is invoked. When the partition dispatcher determines that the running partition must change, the partition dispatcher promotes the heir partition to be the next running partition.



AIR PMK Partition Dispatcher	
Pre-requisite	
	Know the running partition
	Know the heir partition
Function	
	Save the context of the running partition, including the full CPU context, if required.
	Clock tick accounting.
	If required, restore the heir partition and CPU execution contexts.

Figure 14 – AIR PMK Partition Dispatcher functionality

Saving/restoring the partition execution context is of fundamental importance to ensure the consistency of RTOS operation inside a partition. That means, the execution contexts of both partition and RTOS running process will be consistently up to date when, in the course of a future partition switching, the same process returns to the running state. This includes clock tick accounting.

4.2.3 Process Scheduling and Dispatching

After execution of AIR PMK partition dispatching actions, the system clock tick ISR invokes the native RTOS functions for process scheduling and dispatching. This includes clock tick processing, which is restricted to the scope of the active partition. Thus, the temporal interference between partitions is minimal with this regard. Clock tick processing ensures that any process becoming active will be placed in the RTOS queue of ready processes[4] upon partition switching.

No fundamental changes are needed to RTOS components, exception made to the module performing clock tick processing.

[4] A process in the ready state has all the resources required to be executed but the processor.

5 AIR APEX Interface Implementation

This section discusses the implementation of the ARINC 653 APEX functionality onto RTEMS and how it should be integrated in the AIR architecture.

The current APEX interface implementation is tightly coupled to the proof of concept demonstrator, described in section 8.3 of this document. The demonstrator is centered on the interaction of a number of processes within a single partition. Therefore this implementation composes, in this scenario, a relevant sub-set of the overall APEX functionality, including the APEX process and partition management services and a number of intrapartition services.

As the AIR project has evolved, its system architecture has considerably grown to be modular and highly flexible. It now encompasses the notion of a **multi-executive** system with a different instance of the same RTOS per partition, and it is ultimately foreseen to encompass a multi-executive system that allows the usage of any executive (RTOS) per partition. As a consequence, the capability of providing an APEX as independent as possible from the specificities of a given RTOS kernel was also foreseen.

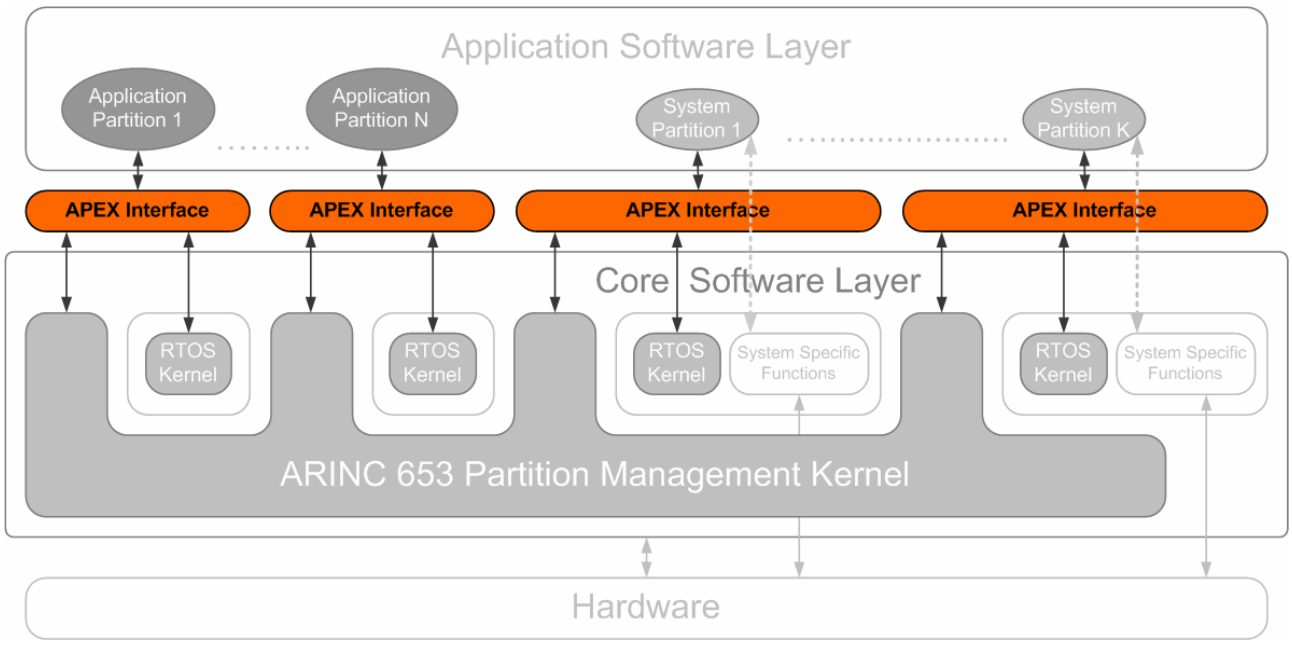


Figure 15 – The APEX interface in the multi-executive AIR architecture

As important as this choice would be regarding the **nature of the APEX implementation**, the AIR team considered the option for using [RTEMSPOSIX], hence providing the desirable portability of the resulting APEX into a given POSIX compliant RTOS. However, an evaluation of the problem indicated that such an implementation could lead to a more complex implementation and added functionality wrapping, as well as raise qualification issues [RTEMSQUAL].

The chosen approach is implemented mostly over RTEMS native API [RTEMSAPI] but still uses the implementation of POSIX modules where it was found pertinent and cost-effective. The team sees the current work as a stepping stone to achieve a POSIX APEX.

The next sub-sections discuss the development environment and the APEX interface modules considered in this implementation.

5.1 Development Environment

The APEX interface development environment consisted on a set of interacting Open Source software tools, deployed on an Intel x86-based desktop PC.

TOOLS	Description	Role
Eclipse IDE	Fedora Core Release	software development environment
rtems GCC	OAR corporation gcc-3.2.3-20040420	rtems backend GCC compiler
Fedora Core 6	Linux based OS	host OS
RTEMS	4.6.6 Release	target OS

Table 7 – APEX Interface development environment

5.1.1 Initialization

Initialization is the moment when partition creates all objects before starts actually executing. This process is accomplished by RTEMS init task, which prepares a simulated partition environment so APEX could have initiate mode as seemed as possible with PMK.

In order to create an abstract context of development, specific places were created for APEX users to define their own processes and initialization code, so they don't have to worry about how it is working. Therefore, a user must create an **`user_init_code`** function and include **`USER_InitializationCode.h`** to define the initialization code. The process entry points can be in the same file as **`init`** as well as in any different file.

```
#include "USER_InitializationCode.h"
/* other additional includes needed */

void user_init_code()
{
    PROCESS_ID_TYPE process1_ID;
    BLACKBOARD_ID_TYPE blackboard1_ID;
    RETURN_CODE_TYPE return_code;

    process1Atribs.BASE_PRIORITY = 30;
    process1Atribs.DEADLINE = SOFT;
    strcpy(process1Atribs.NAME,"P1");
    process1Atribs.PERIOD = INFINITE_TIME_VALUE;
    process1Atribs.STACK_SIZE = 30;
    process1Atribs.ENTRY_POINT = &process1Code;
    process1Atribs.TIME_CAPACITY = INFINITE_TIME_VALUE;

    CREATE_PROCESS ( /*in */ &process1Atribs, // process attributes
                     /*out*/ &process1_ID,    // process id
                     /*out*/ &return_code );

    CREATE_BLACKBOARD ( /*in */ "BB_P2",        // name
                        /*in */ 30,            // max message size
                        /*out*/ &blackboard1_ID, // blackboard id
                        /*out*/ &return_code );
}
```

Figure 16 – init CODE

5.2 Implemented APEX Services

This section describes the APEX interface modules and related sub-set of services implemented for the proof of concept demonstrator.

5.2.1 AIR Partition Management

Partition management module does not exactly match the ARINC 653 **partition concept**. It effectively manages partition information and data, but the *real weight* of enforcing time and space segregation is left to the PMK modules.

The Partition Management module, works as an interface between PMK modules and the APEX modules that require its specific functions. Internally it shall only maintain a **`PARTITION_STATUS`** variable. Partition Management must access its *internal* objects (processes, semaphores, blackboards, events, buffers, sampling and queuing ports) so it can manipulate the processes' data or destroy all resources, upon shutting down a partition.

Implemented Services

`SET_PARTITION_MODE` is implemented and used in the proof of concept demonstrator initialization phase. It currently supports setting the partition mode to **`NORMAL`**.

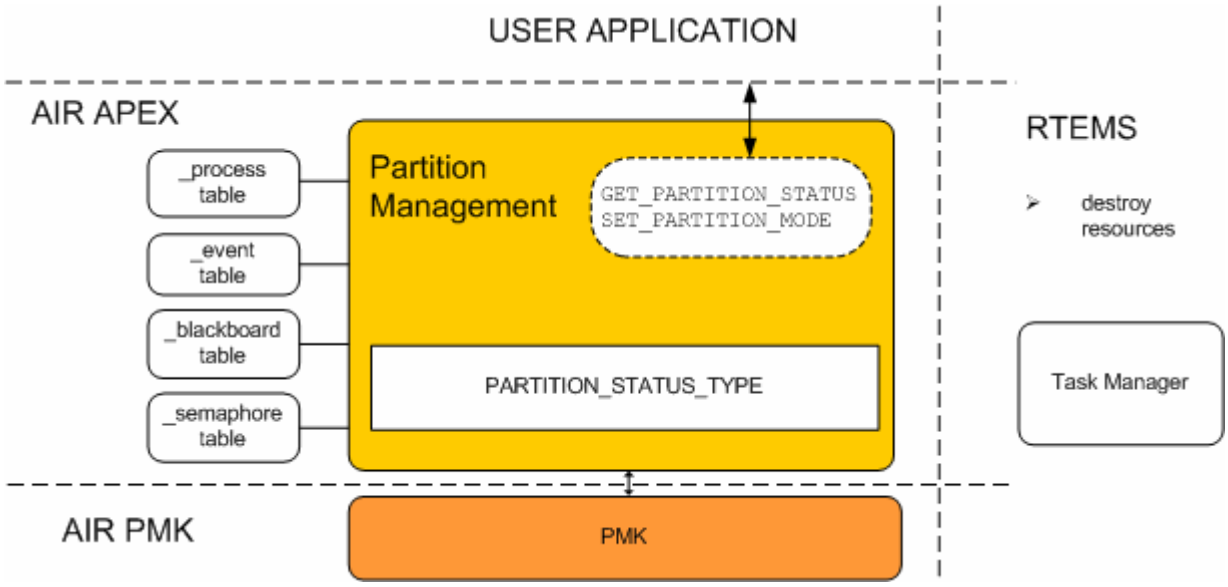


Figure 17 - Partition Management interfaces

5.2.2 AIR Process Management

Process management encompasses key APEX functionality concepts that are defined in terms of an already existing system, in this case the RTEMS core and its native API [RTEMSAPI].

AIR Process definition

The **ARINC 653 process** definition is implemented as a RTEMS task, as it is defined in RTEMS native API. ARINC 653 describes a process as being one programming unit contained within a partition which executes concurrently with other processes of the same partition; it comprises the executable program, data and stack areas, program counter, stack pointer, and other attributes such as priority. All processes are uniquely identifiable, having attributes that affect scheduling, synchronization, and overall execution.

A **RTEMS task** is a thread of execution that competes on its own for system resources and is manifested by the existence of a task control block that maintains the control information required by its operation such as the task name and id, its current priority or context.

We call an **AIR process** to the entity joining both definitions. An AIR process is also manifested by the existence of its own control block (process control data) and will always (after it is created) have a corresponding RTEMS task attached to it. Making this correspondence functional and coherent, while strictly observing the ARINC 653 demanded functionality, sums up this module's implementation.

Process control data materializes the AIR process. It includes the process **STATUS**, the type containing both its static and dynamic attributes, and additionally the needed per process control information required to the overall operation of the process management module.

```
typedef struct
{
      // Process Control Data
      PROCESS_ID_TYPE   id;                    //APEX id of a process
      PROCESS_STATUS_TYPE status;              //status of a process

      // timer id for the timer associated to this process
      //rtems_id timer_id;

      // auxiliary flags
      // Boolean timer_expired;                // did the timer expire?
      Boolean stopped;                         // STOP?
      Boolean suspended;                       // SUSPEND?
      Boolean self_suspended;                  // SUSPEND_SELF?

} AIR_Process;
```

Figure 18 – AIR Process Control Data

AIR Process scheduling

The ARINC 653 standard imposes a **preemptive priority-based scheduling** algorithm which ensures that the process which is executing on the processor at any point in time is the one with the highest priority among all processes in the ready state with FIFO order applied to processes with the same priority.

RTEMS own scheduling concepts do match the APEX scheduling rules. The followed approach uses one instance of the RTEMS scheduler for process scheduling inside each partition. **No fundamental**

modification is needed to the functionality of this component for its integration in the AIR system architecture.

RTEMS provides 255 priority levels, corresponding the level 255 to the lowest priority and level 1 to the highest. Opposingly, the APEX specification not only supports a maximum number of 63 priorities, but also makes level 1 correspond to its lowest priority and level 63 to its highest. ARINC 653 specifies, opposingly, that the process *in the ready state with the highest* (most positive) *current priority is always executing while the partition is active*.

	BOUNDS	LOWEST PRIORITY	HIGHEST PRIORITY
ARINC 653	[1-63]	1	63
RTEMS	[1-255]	255	1

Table 8 – ARINC 653 and RTEMS process priorities

The two systems are made correspondent through a simple operation whenever priority values are either read through **`GET_PROCESS_STATUS`** or written, upon process creation or through calling **`SET_PRIORITY`**.

AIR Process state

The states of ARINC 653 processes and RTEMS tasks do not exactly match. There exist in RTEMS no transition to the **`DORMANT`** state (RTEMS has no **`STOP`** directives). These transitions are thus emulated into matching the functionality specified for the APEX.

In order to comply with the specification, the process state should be consistent with its corresponding task state, at any given time, from the point of view of a user application (through invocation of the **`GET_PROCESS_STATUS`** service).

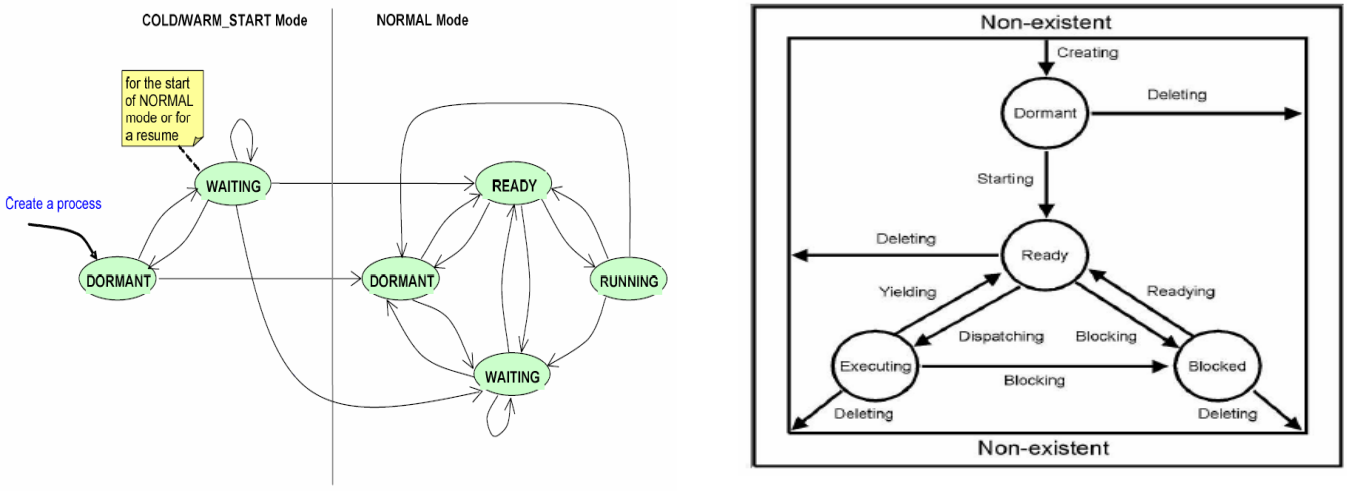


Figure 19 – ARINC 653 process states (on the left) vs RTEMS task states

(NOTE: figures adapted from [A563P2] and [RTEMSAPI])

Also, the **WAITING** and **BLOCKED** states do not strictly correspond, although this is not visible from Figure 19. ARINC 653 details that a process is in **WAITING** state if one or both of the following conditions are applicable:

- process is waiting on a resource: delay, semaphore, period, event, message or for the start of the partition's NORMAL mode; and / or
- the process is suspended: waiting for resume, either after **SUSPEND** or **SUSPEND_SELF** calls.

On the RTEMS side, the **BLOCKED** task state is spread into several other suspension levels / states.

The followed solution is a mixed one: upon a perceivable process state change the **PROCESS_STATE** in AIR process control data is always updated. When this is not possible, the RTEMS Task Control Block (TCB) is directly accessed to obtain the current state of the corresponding RTEMS task. As such from the combination of the information present on the AIR Process structure and the information present on the RTEMS TCB the current AIR process state can always be retrieved.

AIR Periodic Processes

The approach to accomplish the functionality of periodic processes indicated for the usage of the rate monotonic manager.

However we found that using a rate monotonic period would immediately start it upon its creation. This would withdraw the necessary scheduling control derived from the APEX specification that states that the release point of a periodic process shall start on the next partition period.

Due to the great complexity and effort cost foreseen for any effective solution and its little relevance on the current proof of concept work objectives, the specification and implementation of Periodic Processes in the scope of AIR was not considered. In future activities this issue is considered a critical one that must be carefully evaluated as to assure a full APEX interface implementation over RTEMS.

Implemented Services

```
CREATE_PROCESS
START
STOP
SUSPEND
RESUME
GET_MY_ID
GET_PROCESS_ID
GET_PROCESS_STATUS
```

Here above is the relevant sub-set of the implemented services both those that are relevant to cover the process control needs as also those that are used in the proof of concept demonstrator.

5.2.3 AIR Blackboard services

Blackboard services as specified on ARINC 653 standard provide functionalities for intra-partition communication. In contrast to buffer, blackboard does not support a message queue: when a message is displayed it overwrites the previous one. This unique shared message can be cleared at any time by any process on the same partition (in a similar nature to the functionality of sampling ports).

If a message is available at the blackboard, then any process can access it instantly with no reserve. Processes only block on a blackboard when trying to read it while the blackboard is empty. When it becomes not empty, processes are automatically unblocked.

Since there is no similar RTEMS object to which the APEX Blackboard can be mapped to directly, this object shall be defined using a global variable (that shall hold the blackboard message) with access controlled by **mutexes and conditions**. As such a structure was defined containing these fields (Figure 20).

To prevent any concurrency problems, the access to the **`EMPTY_INDICATOR`**, which tells if a blackboard is either empty or occupied, and the access to the blackboard message area must be controlled by *mutexes*. These implement mutual exclusion mechanisms that do not allow two processes accessing the blackboard at the same time. *Conditions* shall be used with *mutexes* to block readers when the board is empty and notify everyone (with **`pthread_cond_broadcast`**) once it changes to *occupied*.

```
typedef struct
{
      /** Blackboard Control Data **/

      BLACKBOARD_ID_TYPE      id;         //APEX id of a blackboard
      BLACKBOARD_NAME_TYPE    name;       //APEX name of a blackboard
      BLACKBOARD_STATUS_TYPE  status;     //status of a blackboard

      //length of the last displayed message on blackboard
      MESSAGE_SIZE_TYPE length;

      // mutual exclusion control, to write to blackboard
      pthread_mutex_t mutex;

      // condition to grant access to mutual exclusion
      pthread_cond_t cond;

      // blackboard shared memory
      unsigned char shared_mem[SYSTEM_LIMIT_MESSAGE_SIZE];

} AIR_Blackboard;
```

Figure 20 – AIR Blackboard Control Data

Implemented services

CREATE_BLACKBOARD
DISPLAY_BLACKBOARD
READ_BLACKBOARD
CLEAR_BLACKBOARD
GET_BLACKBOARD_ID
GET_BLACKBOARD_STATUS

All the Blackboard service requests are used in the proof of concept demonstrator, allowing for the communication of processes within a partition. These services are also representative of the overall intra partition services.

6 AIR RTEMS Integration

Aside from the components that need to be implemented from scratch that are limited to the AIR Partition Management Kernel (PMK), the AIR architecture uses the services provided by the RTOS kernels. These services provide not only the underneath scheduling policies but also a set of comprehensive functionality needed for the integration of the APEX interface.

In the RTEMS case, a set of managers can be selected to provide this mapping to the APEX interface, accordingly to the necessity. Apart from the RTEMS managers, there are also other RTEMS components that integrate the AIR architecture. Such components are used, for example, for AIR Hardware Abstraction Layer (HAL) initialization, device driver implementation, standard interfaces, etc. However, each one of these modules has a specific task that in some cases can not be replicated throughout the partitions. These are mainly the components that address the hardware and are restricted to either system partitions or to the AIR Partition Management Kernel (PMK). In Table 9 on next page is depicted the RTEMS Managers and other components used by the AIR design solution and their localization within the overall architecture.

The blank cells in Table 9 indicate that the component is not present in the architecture but that no design decision to include/exclude that particular component was required. A red cross symbol (✗) explicitly rules out the component supported on a design decision process while a green check (✓) symbol includes the component in the specified module of the AIR architecture.

The diagram in Figure 21 summarizes the RTEMS modules that have been excluded from partition operation by design considerations. For example, the RTEMS modules implementing dynamic memory allocation have been excluded to preserve determinism and real-time operation. On the other hand, the RTEMS interrupt manager was excluded from partition modules because a similar functionality is provided by the AIR PMK.

Similar considerations do apply to other modules, in the rightmost diagram of Figure 21. The functionality of the excluded modules is either provided by the AIR PMK or is included specifically in the system partitions. An exception made to the Itron Interface manager that has been simple excluded. Some of the canonical build tools are used for specific AIR functions (e.g. in the implementation of object file filtering of the naming structure).

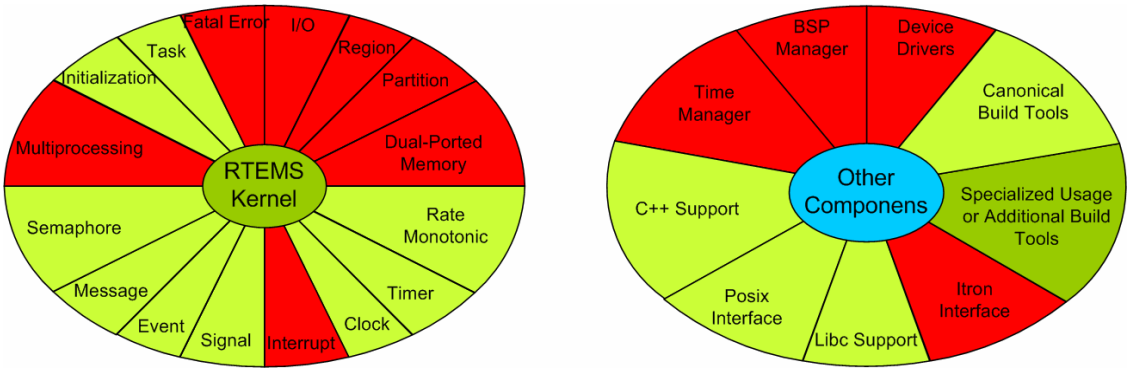


Figure 21 – RTEMS modules excluded from partitions by AIR design

RTEMS Kernel			AIR PMK Managers
Manager	**In all Partitions**	**Only in System Partitions**	
Initialization Manager	✓		
Shutdown Manager	✓		
Task Manager	✓		
Semaphore Manager	✓		
I/O Manager	✗	✓	✓ (optional)
Interrupt Manager	✗	✗	✓
Partition Manager	✗	✗	✗
Region Manager	✗	✗	✗
Clock Manager	✓		
Timer Manager	✓		
Message Manager	✓		
Event Manager	✓ (optional)		
Signal Manager	✓ (optional)		
Dual-Ported Manager	✗	✗	✗
Fatal Error Manager	✗	✗	✗
User Extension Manager	✗	✗	✗
Multiprocessing Manager	✗	✗	✓ (optional)
Rate Monotonic Manager	✓ (optional)	✗	✗
Other Components			
RTEMS BSP Manager	✗	✗	✓ AIR HAL (mandatory)
Device Drivers	✗	✓	✓ (optional)
POSIX Interface	✓ (optional)		
µItron Interface			
Libc support	✓ (optional)		✓ (optional)
C++ support	✓ (optional)		
Time Manager	✗	✗	✓
Canonical Build Tools	✓	✓	✓
Specialized Usage of Canonical Build Tools			✓

Table 9 - RTEMS Components and their integration in the AIR Architecture

7 AIR Application Build Process

This section discusses a set of relevant issues concerning the build process of a AIR Multi-Executive Core application.

7.1 Build Tools and its Utilization

The development tools used in the development of an AIR-based system are the standard GNU development tools, as used in the development of standard RTEMS applications.

However, thought the RTOS kernel objects to be deployed in different partitions should be regarded as different objects they use the same naming references. Thus a methodology is needed to handle this problem. A filter is used to modify the native RTEMS naming structure with regard to each public symbol in a given partition application file. The application files of each partition may afterwards be combined by a canonical link editor into a single application file (cf. Figure 22).

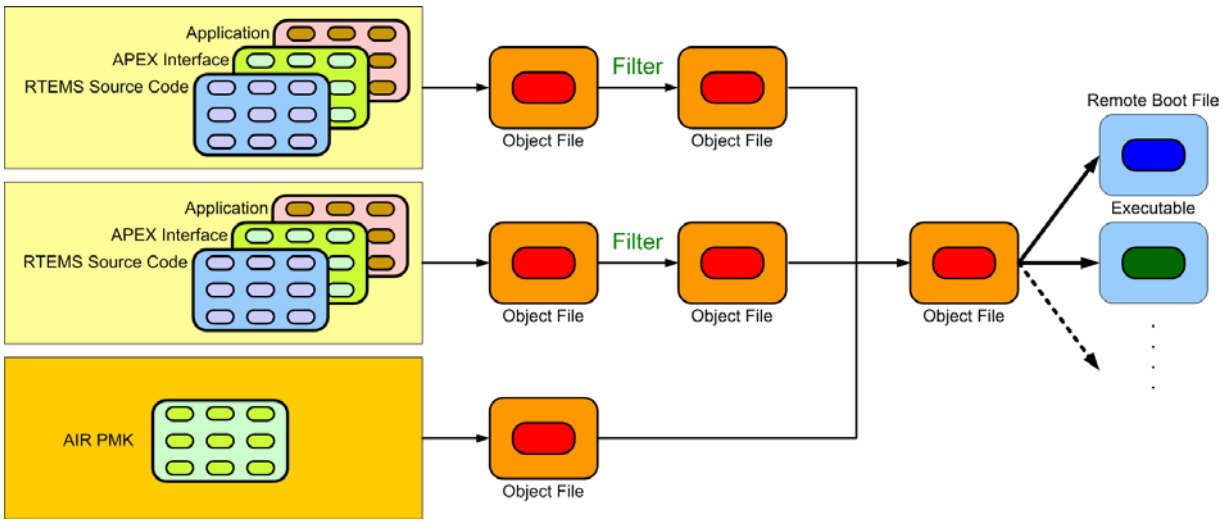


Figure 22 – Specialized Use of Canonical Build Tools in Multi-Executive Core Development

The use of a single application download file is assumed in the design of the standard bootstrap methodologies such as those used by the GRUB tool, included in the GNU package distribution.

The use of a single application download file is also almost mandatory when GNU-based debugging tools, such as **gdb** (GNU debugger) and **ddd** (data display debugger) are used with synthetic processor targets, such as the SPARC Instruction Simulator (SIS) used earlier with SPARC ERC32 processor platforms and/or the recent TSIM simulator for the radiation-hardened fault-tolerant SPARC LEON architectures.

8 AIR Proof of Concept Prototypes

This section briefly describes the proof of concept prototypes developed for and presented in the final review of the AIR Project.

8.1 Prototyping Environment

The proof of concept prototypes were built using the RTEMS 4.6.6 version enhanced with a graphical window manager, dubbed VITRAL (VITRAL is the Portuguese word for stained glass window) developed within other projects [COUT06a] but also considered useful in the context of the AIR Project. The utilization of VITRAL in the context of the AIR PMK proof of concept demonstrator is illustrated in Figure 23.

Figure 23 – The VITRAL visualization environment as used in the AIR proof of concept prototypes

The VITRAL window manager is built for real-time embedded systems. As such, it has little memory footprints and time-bounded functionality. The hardware utilization requirements are minimal: in the essence it merely requires a support for a standard VGA video interface.

Standard distributions of the RTEMS kernel only have a generic console for user interaction. In that console, there is only one window to which all the tasks must perform their keyboard input/screen output. Hence, the output of multiple tasks is many times confusing and is not user friendly. Moreover, the standard console does not present fundamental real-time properties: tasks waiting for a key perform active waits.

Input/Output interaction adds an unknown time interference variable to the remaining system. With regard to the keyboard, pressed keys trigger interrupts that postpone the executing task. Thus, VITRAL is equipped with a “state of the art” component that prevents unbounded interrupt interference [COUT05a].

Using the standard console, the application typically interfaces with the OS and underlying hardware through the standard I/O library - the ANSI C Library. While the primitives provided by this library are sufficient for text based applications, window management is not incorporated. A programming extension (the VITRAL API) is needed to provide support for application development. Figure 24 illustrates the integration of the OS and VITRAL with the application, while maintaining the existent ANSI C functionality.

The application interfaces with the standard I/O library to perform regular I/O functions, such as a `printf` or a `scanf` call, and accesses window management through the VITRAL API. Through a portability layer, the VITRAL core is capable of interfacing with several distinct OS and devices, as long as they provide some basic functionality. Thus, VITRAL offers an enhanced multicolor text mode window manager while maintaining a low resource usage (~ 60 KiB) and is designed for real-time systems [COUT06a]. VITRAL is built under a highly modular architecture which makes it easily portable to other Operating System kernels or underlying hardware as described in [COUT06a].

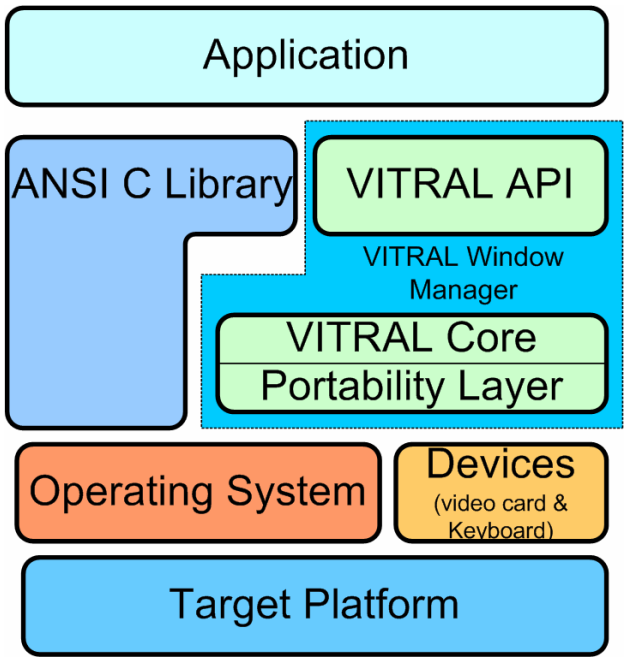


Figure 24 – Architecture of the VITRAL visualization environment

8.2 Multi-Executive Core Demonstrator

This section briefly presents the structure of the proof of concept demonstrator for the AIR Multi-Executive Core architecture.

The AIR Multi-Executive Core (MCE) proof of concept demonstrator illustrates partition and task scheduling using multiple RTEMS kernels (one per partition and another for the support of the AIR PMK Partition Manager). A possible assignment of functional components to partitions in a space application is exemplified in the table of Figure 25. The scheduling of those partitions over a MTF (Major Time Frame) is also illustrated in Figure 25. Each partition is composed by a set of periodic tasks which print a string to a VITRAL window, as shown Figure 23.

Partition	Function
Partition P1	Attitude Control
Partition P2	Telemetry Tracking and Command
Partition P3	On-Board Data Handling
System Partition	Communications

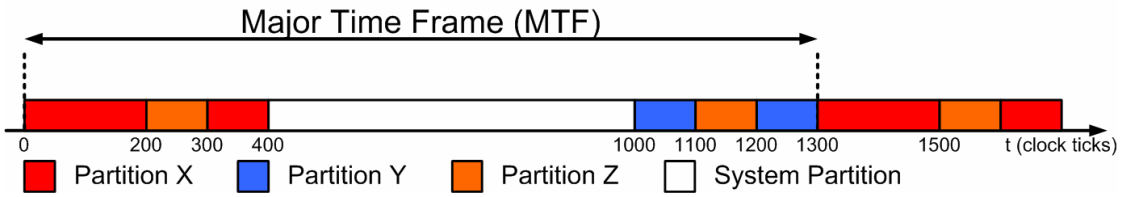


Figure 25 - Example of Partition function assignment and scheduling over a Major Time Frame

8.3 APEX Interface Demonstrator

This section thoroughly discusses and presents the demonstrator of the proof of concept implementation of the APEX interface on AIR.

The demonstrator does not aim to demonstrate the total compliance of AIR implementation with the ARINC 653 standard or to exhaustively test its correct functionality. Instead, the purpose of this demonstrator is to show evidence that basic APEX features can be implemented over RTEMS and that **correct** behaviour is achieved. As such it provides a valid evidence to advance for a full APEX implementation over RTEMS.

The demonstrator main focus relies on the following issues:

- Basic operations over APEX processes like **`CREATE`**, **`START`**, **`STOP`**, **`SUSPEND`**, **`RESUME`** using RTEMS tasks to implement APEX processes;
- Intra partition communication between AIR processes by using the APEX Blackboard service;
- The implementation of the APEX priority driven process scheduler by using the RTEMS native task scheduler

We also would like to note that the APEX Interface Demonstrator works only by itself over the RTEMS O/S. This means no integration with the PMK was performed as this was not a relevant issue, since the PMK proof of concept was completely independent of the APEX Implementation over RTEMS proof of concept.

As such where the APEX Interface Demonstrator has identified dependencies over the PMK (partition management and inter partition communication), stub functions are used as to simulate the PMK functionalities.

Next we provide a description of the resources used by the APEX Interface Demonstrator followed by a description of its functionality.

8.3.1 APEX interfaces demonstrator resources

As to implement the APEX Interface Demonstrator the following APEX resources were used:

- **Process P1** – this is the main process that sends messages to the blackboards. It has the lowest priority of the three processes used;
- **Process P2** – this process waits for a message to be available on blackboard BB2. When this condition is met, it reads the message and clears the blackboard. It then reacts according with the received message. Its priority his above P1.
- **Process P3** – this process is similar to P2 but uses blackboard BB3 instead of blackboard BB2. Also, it has the same priority as P2.
- **Blackboard BB2** – this blackboard shall be used for P1 to send messages to P2;
- **Blackboard BB3** – this blackboard shall be used for P1 to send messages to P3.

8.3.2 APEX Interface Demonstrator functionality

The start up task

As defined on ARINC 653, each partition has associated an entry point that performs all start up code and ends by setting the partition to mode **NORMAL**. The APEX Interface demonstrator works in the same manner as to be compliant with the ARINC 653 standard. However because the PMK is not integrated on the simulator the calls to it (for partition mode change and to get partition status) are simulated.

Given this the partition start up task will create all the resources needed to the partition, which in this case are the 3 processes and the two blackboards. The start task shall also start up process P1 which shall remain waiting while partition does not enter de **NORMAL** mode.

Next we present the pseudo code for the partition start task.

```
CREATE_PROCESS (P1)
CREATE_PROCESS (P2)
CREATE_PROCESS (P3)

CREATE_BLACKBOARD (BB2)
CREATE_BLACKBOARD (BB3)

START P1

SET_PARTITION_MODE (NORMAL)  --process P1 takes control
```

Figure 26 – Start task pseudo code

P2 and P3 basic functionality

As stated before P2 and P3 will have very similar functionality. Their functionality is based on an infinite loop. Once started the process shall enter the infinite loop. The process shall then try to retrieve a message from their correspondent blackboard which is BB2 or BB3 respectively if process is P2 or P3. Once a message is received, the process shall interpret its content and either perform the request contained on it or discard it as meaningless. Finally the process shall clear the blackboard and again wait for another message from the blackboard.

The request contained on the message always refers actions to perform over the other similar process. This means that if process is P3 the actions shall be done over P2 and the opposite. These actions are either to suspend, to resume or to stop the similar process. Also noticeable is that, if at a given moment both P2 and P3 can execute, then the oldest process waiting to get processor time shall be executed. If one process is running and does some action that makes the other (same priority process) to be executable then the now able to execute process pre-empt the process running. Next we present a resumed pseudo code of the P2 and P3 implementation.

```
WHILE ()

   READ_BLACKBOARD (BBx, INFINITE_TIME_VALUE, message)

   IF ( message == doSomeAction)
      doAction
   ELSE
      discard message
   CLEAR_BLACKBOARD (BBx)

END WHILE
```

Figure 27 – Generic P2 and P3 process pseudo code

P1 basic functionality

Process P1 shall be the first to execute because it shall be the only one started at initialization stage. Its first action shall be to start up P2 and P3. Because both have higher priority than P1 each of them shall pre-empt P1 as soon as they start. However, as explained on the section above, they shall soon block waiting for a message to be available on their blackboards. As such, control is returned to P1. After P2 and P3 are started and blocked waiting for messages on their respective blackboards, P1 shall send a set of messages to the blackboards as to make P2 and P3 execute some actions. These actions will allow the visualization of the APEX interface proof of concept in execution.

Next we present the P1 pseudo code with detailed comments on the effects of each message sent.

```
START (P2)   -- process P2 takes control and blocks waiting for a message on BB2
START (P3)   -- process P2 takes control and blocks waiting for a message on BB3

DISPLAY_BLACKBOARD (BB3, “dumb message”)  -- process P3 takes control, reads blackboard
                                          -- BB3, discard the message, clear blackboard
                                          -- and blocks again

DISPLAY_BLACKBOARD (BB2, "P2 suspend P3")  -- process P2 takes control, reads blackboard BB2
                                           -- suspend process P3, clear blackboard BB2 and
                                           -- blocks again

DISPLAY_BLACKBOARD (BB3 "P3 stop P2")      -- message is not read by P3 because it is
                                           -- suspended

DISPLAY_BLACKBOARD (BB2 "P2 suspend P3")   -- process P2 takes control, reads blackboard
                                           -- BB2, try to suspend process P3 but it is already
                                           -- supend, clear blackboard BB2 and blocks again

DISPLAY_BLACKBOARD (BB2 "P2 resume P3")    -- process P2 takes control, reads blackboard
                                           -- BB2 and resume process P3,
                                           -- process P3 takes control and reads message
                                           -- "P3 stop P2" from blackboard BB3
                                           -- P3 stops P2, clear message from BB3 and
                                           -- blocks again

DISPLAY_BLACKBOARD (BB3 "P3 stop P2")  -- process P3 takes control, reads blackboard BB3
                                       -- and does nothing because P2 is already stopped,
                                       -- clear blackboard BB2 and  blocks again

START (P2)  -- process P2 takes control, reads message "P2 resume P3" from blackboard BB2
            -- which was not cleared before and fails to resume P3 because it is not suspended
            -- clear blackboard BB2 and  blocks again

START (P1)  -- fails to start process P1 because it is already running (it’s himself)

START (P2)  -- fails to start process P2 because it is already running

STOP (P1)   -- fails to stop itself as this is done by STOP_SELF (not implemented)

STOP (P2)   -- stops process P2

STOP (P3)   -- stops process P3
```

Figure 28 – P1 operation pseudo code

8.3.3 APEX Interface Demonstrator execution and results visualization

The APEX Interface Demonstrator is integrated with the VITRAL visualization environment described on section 8.1. Each VITRAL window holds the output from each of the APEX processes used by the APEX Interface Demonstrator. The code and pseudo code identified on the previous sections shall be populated with output messages that shall identify precisely where the process is and what is doing.

On Annex B can be found a set of activity diagrams that detailed describe the APEX Interface Demonstrator behaviour.

9 Conclusions and Future Challenges

This document provides a detailed analysis of the AIR design solutions which has driven the decision for the adoption of a multi-executive two-level hierarchical core.

A specific AIR Partition Management Kernel (PMK) incorporates the fundamental functionality needed for conformity with the ARINC 653 standard specification. A logical connection to a set of RTOS kernel instances provides the remaining functional requirements.

The AIR architecture specification also addresses the fundamental issues concerning the enforcement of spatial segregation, through memory protection mechanisms, and temporal segregation, through a partition scheduling fixed cyclic policy.

Some engineering issues, such as additional and/or new utilization of existing build tools and the identification of how the RTEMS modules need to be incorporated in the design have also been addressed.

Other relevant component, build on top of the RTOS application programming interface is the ARINC 653 standard application executive (APEX) interface.

The mapping task was challenging due to the differences between RTEMS API and the requirements defined in the ARINC 653 standard. The main differences found were:

- tasks states of RTEMS and process states of ARINC 653 do not match;
- ARINC 653 process are always in memory while RTEMS tasks are deleted from memory when stopped;
- the lack of functionality in RTEMS to provide at any given moment the task status (it only has a task stating if it is suspended or not);
- the implementation of the periodic processes using the rate monotonic manager was not possible with the current functionalities of the referred manager.

These differences implied extra analysis time and added complexity on the built specification. As an example the need for AIR to include internal variables as to maintain its process states updated and coherent with the RTEMS task value was a hard task to achieve.

Even with all the referred difficulties the present document provides a very detailed specification of how to implement the ARINC 653 API over RTEMS. One key point is that not all APEX resources can be directly mapped on RTEMS. Either some extra functionality must be added to the RTEMS native resources or the APEX mechanisms must be implemented from scratch.

As such and regarding the intra partition devices:

- Blackboards are implemented from scratch and do not use any RTEMS device to map their functionality.
- Buffers are implemented using the RTEMS message queues but extra functionality was defined as to implement a send message waiting queue.

Regarding the inter partition communication mechanisms the design of the services was done by using:

- Sampling ports – a structure holding the control variables and a memory area to hold the sampling message and as such the resource is implemented from scratch;
- Queuing ports – usage of the buffer services designed on the scope of the intra partition communication mechanisms which, as seen above, are implemented by using the RTEMS messages queue with extra functionality for the send message waiting queue.

Besides the implementation schemes, an interface with the PMK was identified as needed in order to guarantee that messages were actually transferred between partitions. As such the following set of primitives were defined:

- `AIR_PMK_write_sampling_port`
- `AIR_PMK_read_sampling_port`
- `AIR_PMK_send_notify_queuing_port`
- `AIR_PMK_receive_notify_queuing_port`

These primitives (or similar) are only identified as needed and are not fully designed since the process of transferring data between each partition memory space, which is constrained by the spatial segregation mechanism, is an hardware dependent issue which is out of scope of the AIR activity.

Next we present a list of issues that must be carefully assessed before a future full implementation of APEX can be developed and used as a product:

- **Inter partition communication** – besides completely defining the interface between the APEX implementation and the PMK, the low level memory devices needed to achieve spatial segregation are hardware dependent and as such, the design of the inter partition communication must be carefully evaluated for each kind of hardware configuration;

- **Definition of the APEX interface with the PMK** – this is an AIR system issue that was not 100% solved because it was not on the scope of the project. Although an open issue it should be not a big challenge to solve. This issue was identified for Partition Management but in the future other needs may arise;

- **Implementing periodic processes over RTEMS** – the implementation of periodic processes over the RTEMS native API was informally analyzed among the AIR team and it was evident that some issues might need an extra effort to solve as to make it 100% complaint with the APEX specification. This issue must be formally analyzed as to find the correct solution;

- **Health monitoring** – the health monitor was out of the AIR scope but it is no question that it must be present on any RTOS suitable for space systems. As such, sooner or later it must be implemented over AIR. This should be done always having on mind the issue we address next;

- **ARINC 653 in space specificities** – the adoption of the ARINC 653 standard on space, and as such its transition from the aviation to the space market, need a detailed analysis as to understand the inherent differences and specificities. This could imply some changes over the APEX interface that must be reflected over on the AIR system. A very meaningful example of this is, as referred before, the health monitoring;

- **Optimization** – even after having a 100% compliant implementation of the APEX over RTEMS, any implementation should be carefully tested as to avoid bottlenecks on the system and make it as efficient as possible;

- **Certification** – before proceeding to a fully available RTOS suitable for space usage, the AIR system must be certified as to guarantee the demanded safety level.

ANNEX A AIR RESULTS SUMMARY

This annex summarizes the main results produced within the scope of the **AIR Project – ARINC 653 Interface in RTEMS**.

A.1 AIR DELIVERABLE REPORTS

Work Package	Title	Pages
[AIRWP1]	AIR Requirements, Architecture and Services	63
[AIRWP2]	AIR Overall System Specification	144
[AIRWP3]	AIR Design Results and Proof of Concept	70

Table 10 – AIR Deliverable Reports

A.2 AIR PROTOTYPE DEMONSTRATORS

Work Package	Proof of Concept Demonstrator	Project Usefulness
[AIRWP2-P]	AIR PMK Single Executive Core (SEC)	Validation of fundamental AIR design issues
[AIRWP3-P]	AIR PMK Multi-Executive Core (MEC)	Validates the AIR overall system architecture design
[AIRWP3-P]	AIR APEX Interface	A proof of concept implementation of the APEX over the RTEMS API.

Table 11 – AIR Prototype Demonstrators

A.3 AIR CURRENT PUBLICATIONS

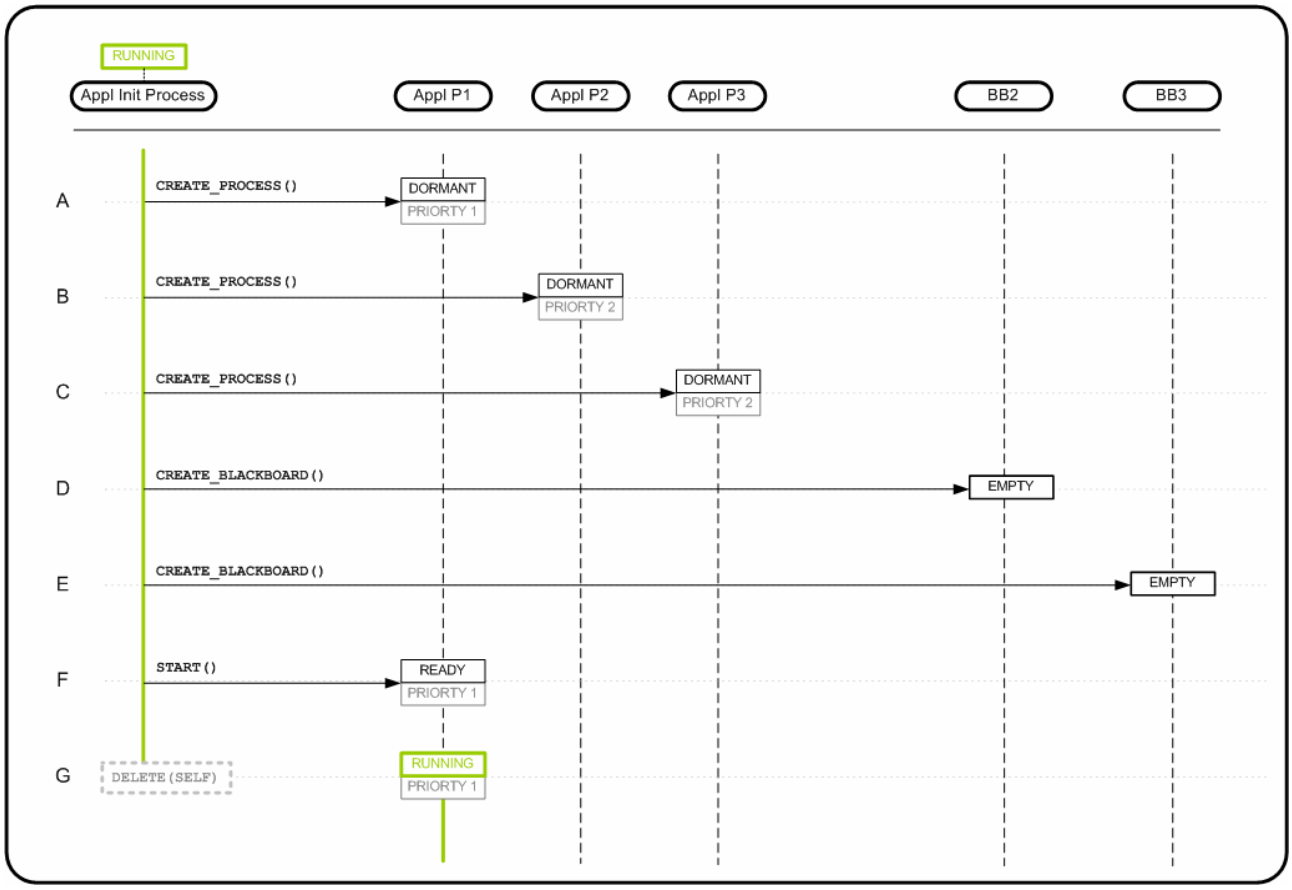
Papers in Conferences or Workshops		
Event	**Title**	**Authors**
DASIA 2007[5]	ARINC 653 Interface in RTEMS	José Rufino, Sérgio Filipe, Manuel Coutinho, Sérgio Santos, James Windsor.

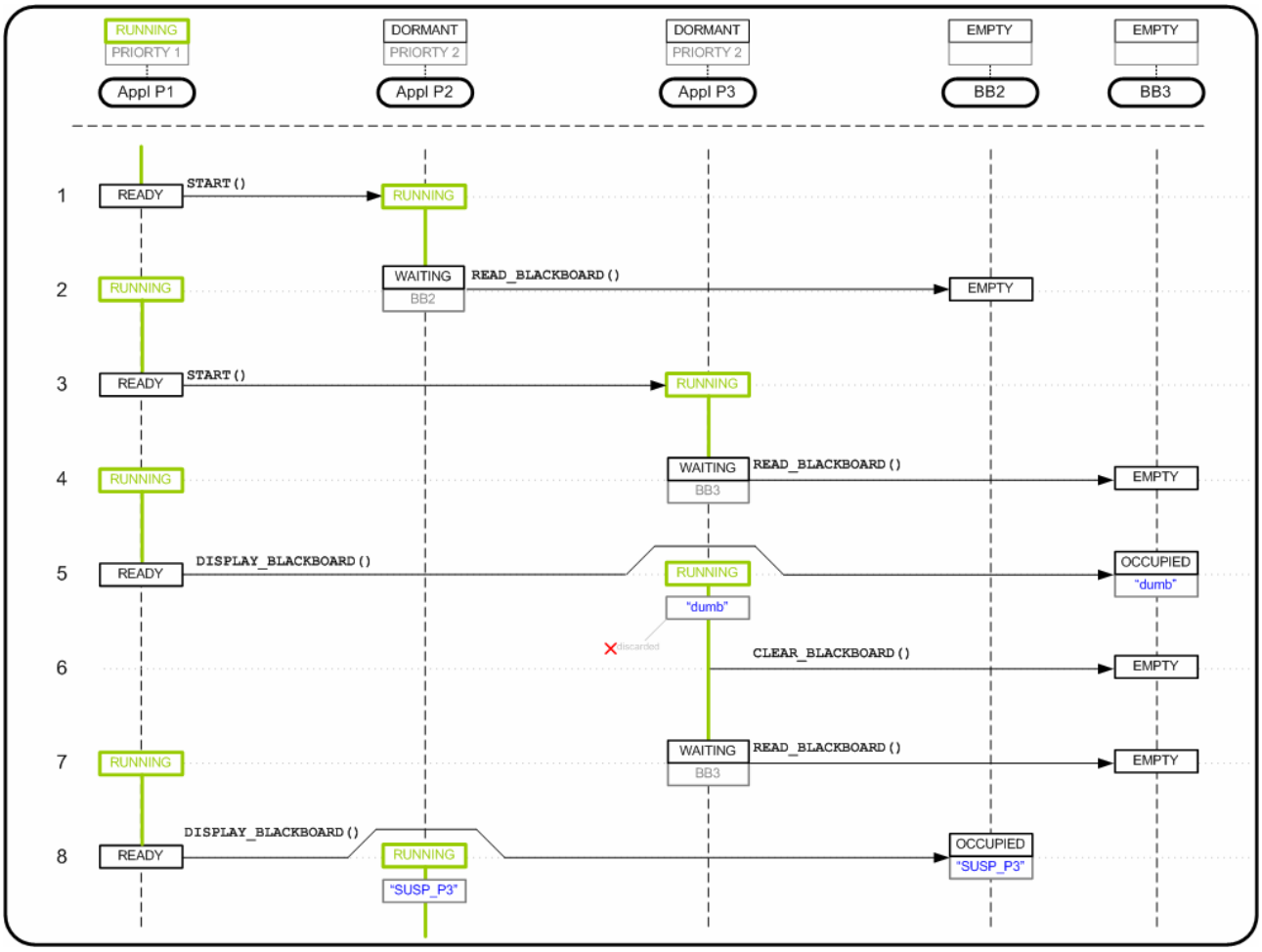
Table 12 – AIR Current Publications

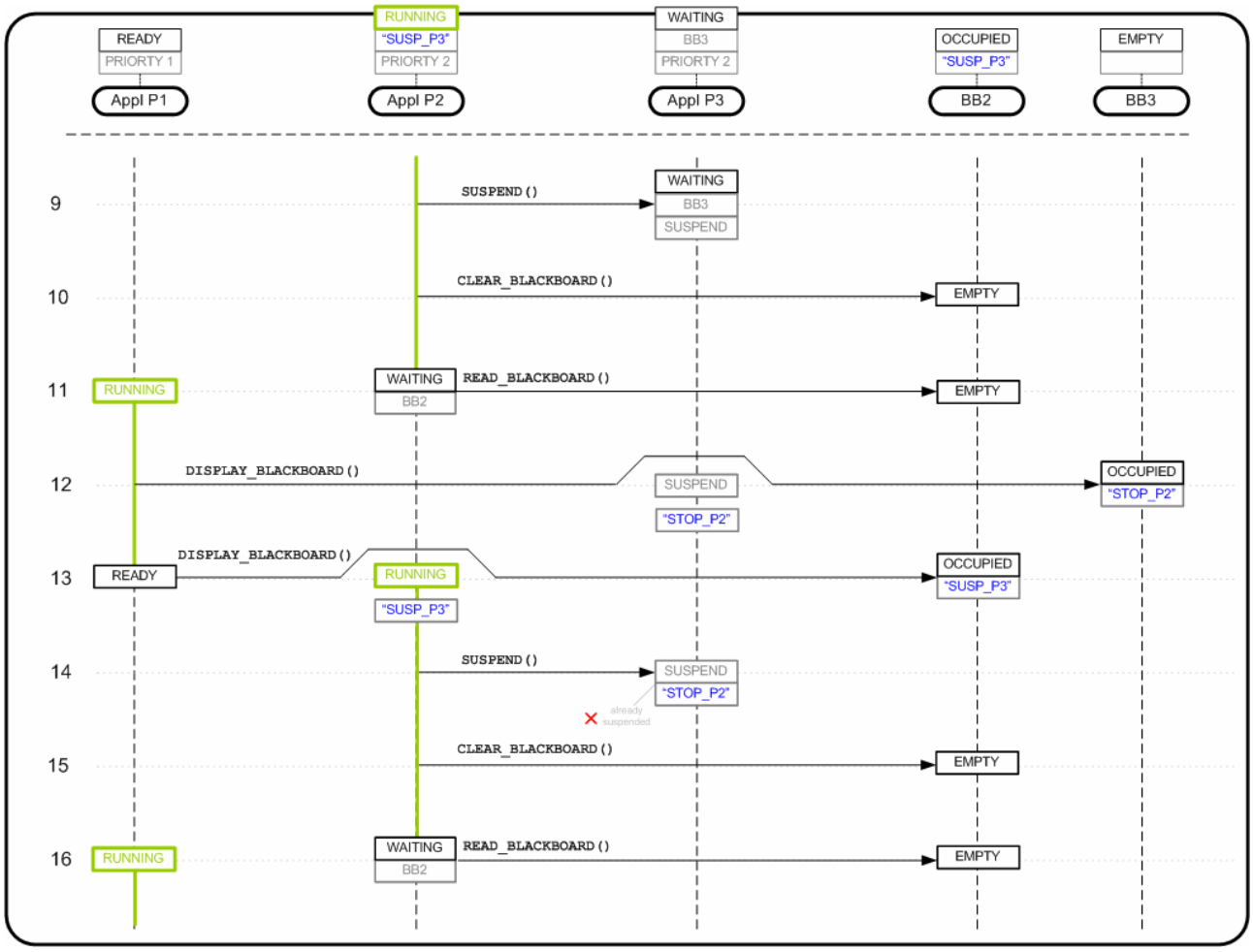
[5] DASIA 2007 – Data Systems in Aerospace, Naples, Italy, May-June 2007

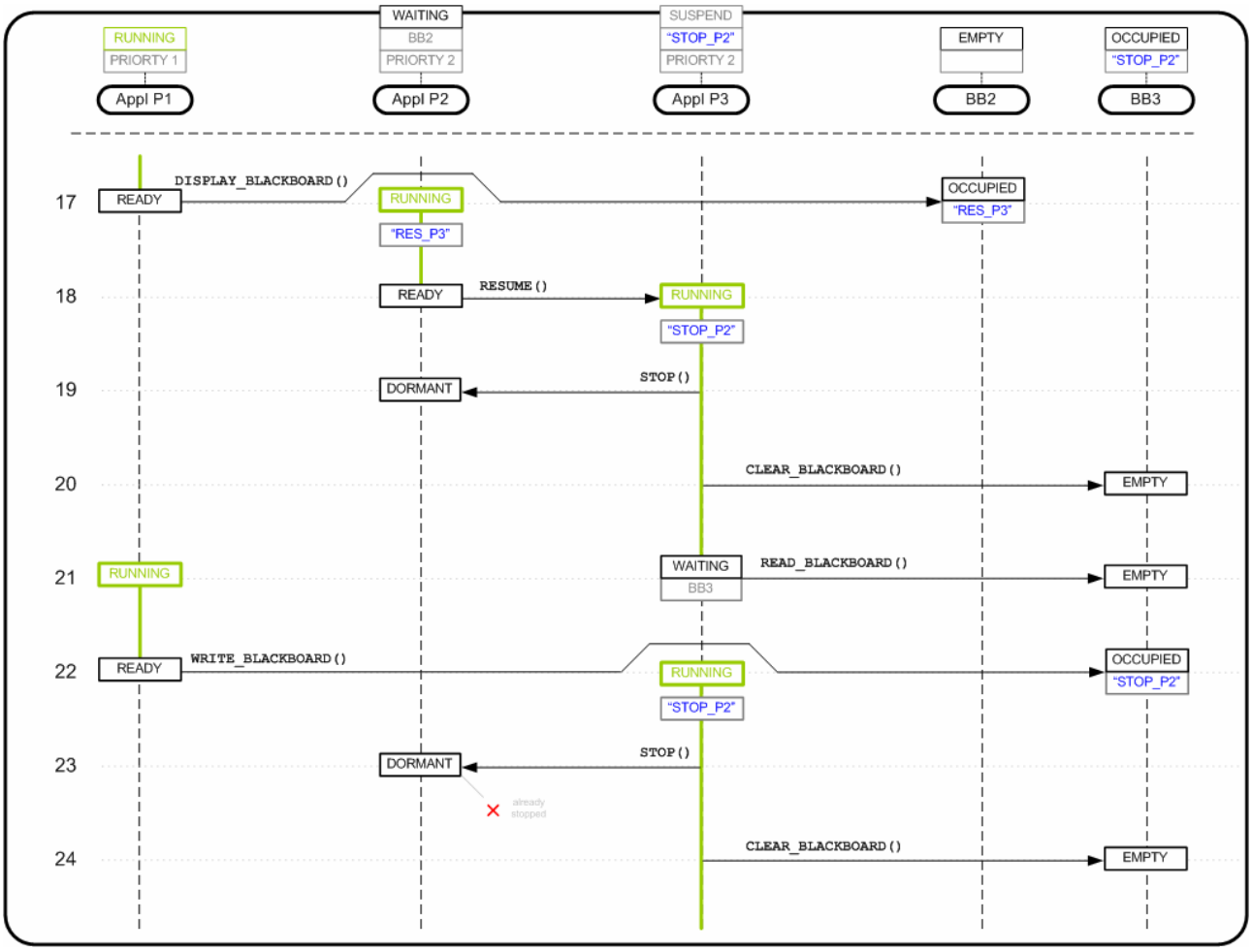
ANNEX B APEX INTERFACE DEMONSTRATOR ACTIVITY DIAGRAMS

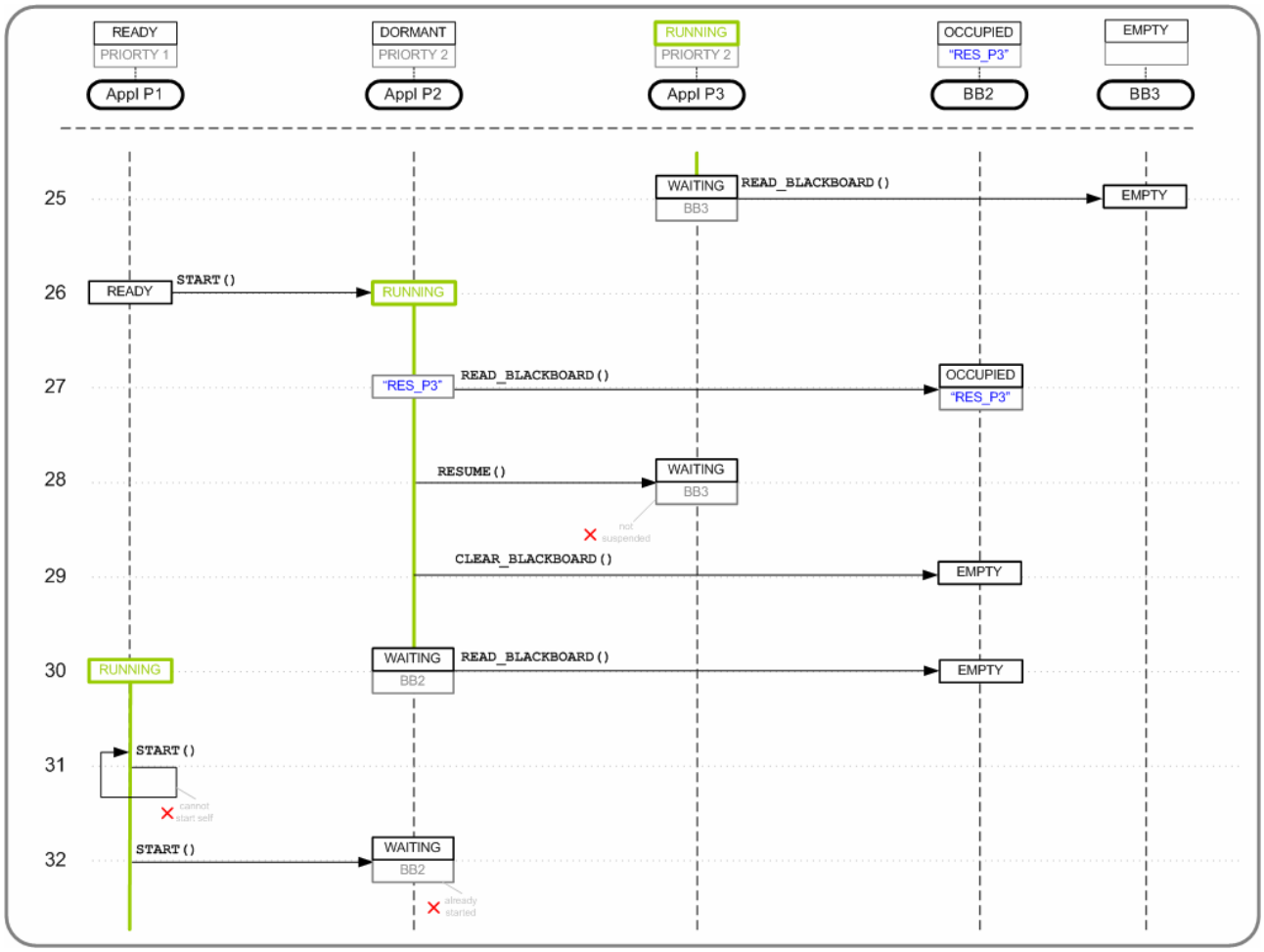
On this annex the activity diagrams of the AIR Interface Demonstrator are presented

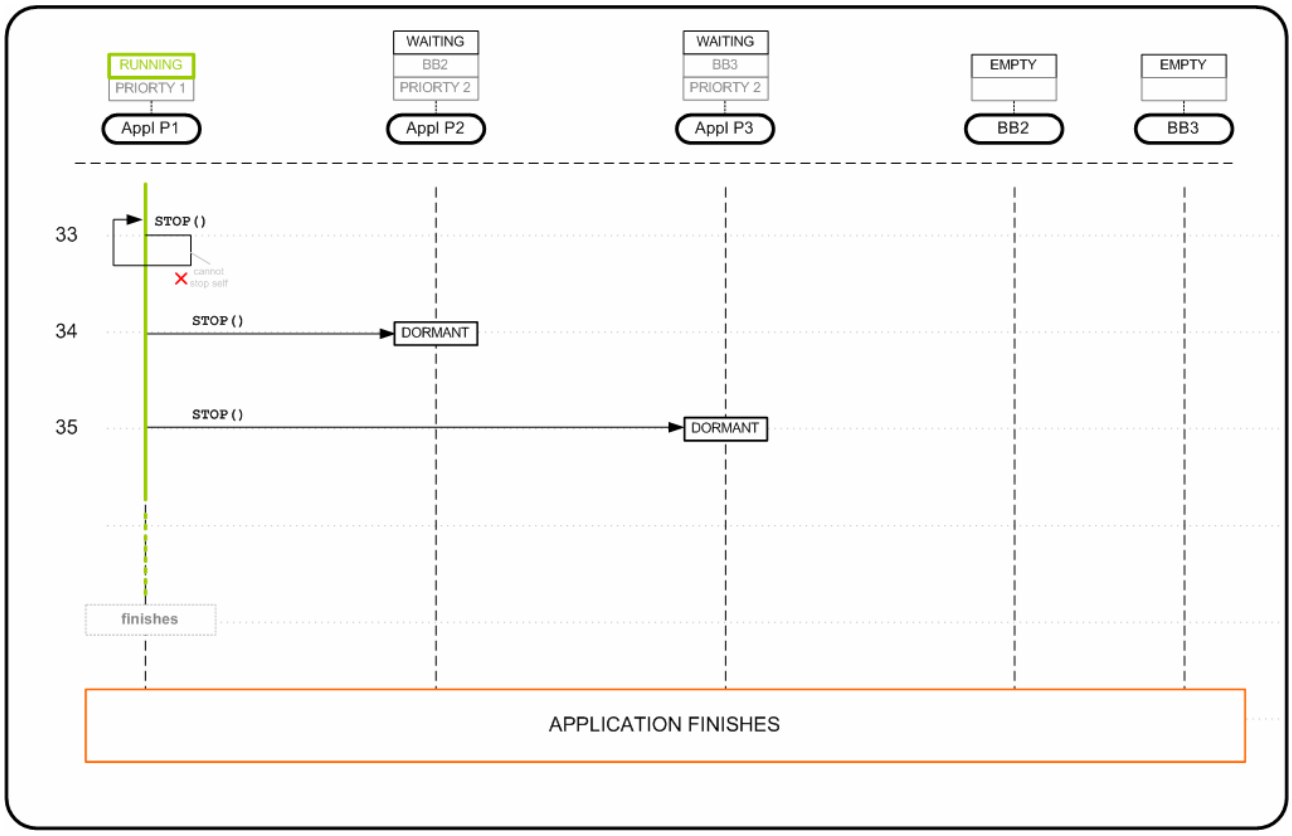











RUNNING
PRIORTY 1
Appl P1
WAITING
BB2
PRIORTY 2
Appl P2
WAITING
BB3
PRIORTY 2
Appl P3
EMPTY
BB2
EMPTY
BB3
33
STOP ()
cannot stop self
34
STOP ()
DORMANT
35
STOP ()
DORMANT
finishes
APPLICATION FINISHES